Fakultät für Informatik
Technische Universität München

Lehrstuhl für Angewandte Softwaretechnik

URML: Towards Visual Negotiation of Complex System Requirements

Florian Schneider

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Alexander Pretschner
Prüfer der Dissertation:	1. Univ.-Prof. Bernd Brügge, Ph.D. 2. Prof. Dr. Martin Glinz Universität Zürich

Die Dissertation wurde am 23.06.2016 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 30.09.2016 angenommen.

Acknowledgements

Before all, I want to thank my thesis advisor *Professor Bernd Brügge, Ph.D.*, who made this thesis possible. He opened up the world of software engineering to me when I was a diploma student, and motivated me to pursue a doctoral degree afterwards. The time I was able to work for his chair was a very enjoyable one, and I learned so much while being research and teaching assistant at his chair. His constructive criticism and the thought-provoking discussions we had throughout my writing time have advanced this thesis such a great deal.

Special thanks also go *Professor Dr. Martin Glinz* who kindly agreed to be the second advisor of this thesis. His work in the RE community has been inspiring and his sharp-witted comments regarding my thesis eminently improved it.

The URML project has been a vision of *Brian Berenbach*, who was a driving force in the research project spawning first versions of the language and getting its reference implementation done. He has been a mentor to me during the project. The project was a reasearch collaboration between Technische Universität München and Siemens in the years 2009-2012. I have to thank *Maximilian Kögel* and *Jonas Helming* for making the contact to Brian and for kickstarting the project in its early phase.

The first reference implementation of URML was created for the UNICASE platform. It was investigated and developed by *Michael Haeger* (Haeger 2011), under supervision of Maximilian and Jonas. This particular implementation helped with early publications on URML (Helming, Kögel, Schneider, et al. 2010; Helming, Kögel, Bruegge, et al. 2010). This early implementation was further extended by Kameliya Terzieva (Terzieva 2011), under the supervision of *Helmut Naughton*. I am particularly indebted to Helmut, not only due to his contributions throughout the project. Thanks for all the fruitful discussions and the good time.

The Enterprise Architect Plug-In was developed at Siemens Corporate Research, Princeton, under the supervision of Brian, Helmut, and me. *Jakob Class* investigated the Enterprise Architect SDK and provided the the first version of the plug-in. Details on his work are in his Master's thesis (Class 2012). *Florian Weigand* worked as an intern at Siemens and provided bug fixes, worked on model migration, and a mechanism for enforcing validity of relationships. The EA plug-in was presented at the 21st International Conference on Requirements Engineering (RE) (Schneider, Bruegge, and Berenbach 2013a). It supported several other publications (Schneider, Naughton, and Berenbach 2012; Berenbach, Schneider, and Naughton 2012; Schneider, Bruegge, and Berenbach 2013b; Berenbach, Rea, and Schneider 2013).

The symbols of the graphical notation were designed by *Christine Kaminski* and *Grace Chi.* Grace worked as an intern at Siemens and refined the initial set of icons developed by Christine. Christine's contribution is well described in her diploma thesis (Kaminski 2011), originally supervised by Jonas, and later by me. *Sebastian Höcht* supported my analysis and comparison of goal-oriented requirements modeling languages. His own judgement of the field can be found in this Master's thesis (Höcht 2012). Thanks also go to all participants of my notation experiment, especially to those fellow doctoral students who gave me some of their precious time.

Widmung

Für Alina, meine Familie, und meine Freunde, die sich sicher freuen werden, wenn ich wieder mehr Zeit für sie habe.

Abstract

During requirements elicitation for complex systems, the requirements analyst has to take a broad range of considerations into account, such as functionality, quality, hazards, threats, features, goals, and workflows. The negotiation of requirements with stakeholders is not only based on the statement of requirements, but also on an understanding of the application domain knowledge.

In order to establish and manage this knowledge, the requirements analyst communicates with various stakeholders to gather their knowledge and transfer it into a consistent requirements model. Iteratively, the analyst presents the model to application domain experts, and updates and extends the requirements model.

Currently no modeling language exists that manages requirements knowledge in a single model. UML only facilitates use case modeling. SysML only extends UML with requirements diagrams. URN supports modeling of stakeholder intentions (goals), but cannot model safety or security issues and product lines. KAOS cannot model features, product lines, and safety or security issues only as obstacles towards the satisfaction of goals. FODA feature diagrams only support the modeling of feature trees.

This dissertation describes the *Unified Requirements Modeling Language (URML)*, a graphical modeling language for requirements engineering. It integrates the requirements knowledge in a single meta-model. In addition, it provides a graphical notation based on Moody's design principles for graphical notation design, with high semantic translucency and semiotic clarity. The applicability of URML to model requirements knowledge has been demonstrated in three case studies. The notation has been empirically validated in an experiment performed with 21 participants.

Zusammenfassung

Während der Anforderungsermittlung für komplexe Systeme muss der Anforderungsanalytiker umfassende Erwägungen wie Funktionalität, Qualität, Gefahren, Bedrohungen, Merkmale, Ziele und Arbeitsabläufe in Betracht ziehen. Der Verhandlung von Anforderungen mit Interessenvertretern liegt nicht nur die Feststellung der Anforderungen zugrunde, sondern auch ein Verständnis der Anwendungsdomäne.

Um dieses Wissen herzustellen und zu pflegen kommuniziert der Anforderungsanalytiker mit verschiedenen Interessenvertretern, um ihr Wissen zu erfassen und in ein konsistentes Anforderungsmodell zu überführen. Wieder und wieder stellt er dieses Modell den Experten der Anwendungsdomäne vor und pflegt und erweitert es dabei.

Zum gegenwärtigen Zeitpunkt existiert keine Modellierungssprache, die es erlaubt, obige Erwägungen in einem Modell zu verwalten. UML ermöglicht die Modellierung von Anwendungsfällen. SysML erweitert UML lediglich durch Anforderungsdiagramme. URN unterstützt die Modellierung von Intentionen (Zielen) von Interessenvertretern, unterstützt aber nicht die Modellierung von Nutzungs- und Informationssicherheit und Produktlinien. KAOS unterstützt keine Produktlinien- und Feature-Modellierung. Sicherheitsfragen werden nur als Hindernisse für eine Zufriedenstellung von Zielen modelliert. Feature-Diagramme in FODA unterstützen lediglich die Modellierung von Feature-Bäumen.

Diese Dissertation beschreibt die Unified Requirements Modeling Language (URML), eine graphische Modellierungssprache für das Requirements Engineering. Sie integriert das Anforderungswissen in einem einzigen Meta-Modell. Zusätzlich liefert sie eine graphische Notation, die auf Moodys Gestaltungsleitsätzen für graphische Notationen basiert. Der Fokus liegt dabei auf den Leitsätzen für 'semantic translucency' und 'semiotic clarity'. Dass URML einsetzbar für die Modellierung von Anforderungswissen im oben genannten Umfang ist, wird in drei Studien aufgezeigt. Die Notation wurde in einem Experiment mit 21 Teilnehmern validiert.

Contents

1. Introduction

Requirements engineering is the activity whose ultimate goal is the formulation of statements that can be passed on to developers to implement a system that satisfies these requirements. Requirements engineering transforms the knowledge distributed over various stakeholder's brains into an externalized model, which we call *early requirements model* (Fig 1.1). The exchange of models and the discussion of shared models is a form of communication (Bruegge and Dutoit 2009, p.21). The information received during communication with stakeholders needs to be reviewed continuously. In order to ensure that the requirements engineers have understood the needs of the stakeholder's organization, which means they usually talk to more than one person, they have to present the knowledge acquired during early requirements work to all stakeholders.

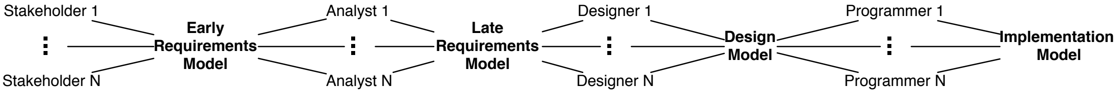


Figure 1.1.: Models as communication artefacts between participants of the development process

Prototypes have been proposed as an appropriate mechanism to establish mutual understanding of how the future system will look like (Ehn 1988). Prototypes can be used to simulate future instantiations of a system design, and thus can help profoundly during the discussion with stakeholders, but if the requirements underlying a prototype's design are not explicated in a model, it will be difficult to validate the prototype. In this dissertation, we focus on the early requirements model. In our point of view, prototypes are a technique complementary to models.

Before requirements information can be passed on to designers, the early model needs to be evaluated and consolidated. Over several iterations, it will be transformed into a late requirements model, which can be discussed with designers. When the late model is stable enough, designers transform it into a design model. The late requirements model already is too technical for the stakeholder. In order to be precise, requirements analysts need to transform early requirements knowledge into terms that have an unambiguous interpretation understandable in terms used by designers, which might also be machine-processable. The closer a model gets to implementable form, the more formal and technical it gets, the farther away from the expertise of the stakeholder. In the same context, Jackson and Zave speak about the transformation of requirements into specifications (Jackson and Zave 1995, p.15). To reiterate in Jackson's and Zave's words, "a specification describes machine behavior sufficient to achieve the requirement". While requirements may refer to domain knowledge important to customers but inaccessible to the system to be built, specifications can only be expressed in terms of phenomena that are visible to that system.

Much research effort has been devoted to the automation of this transformation process and the formalization of its end product. Without precise, correct, and consistent requirements, it is hard to say whether what is required adequately expresses the needs of customers. And without precise, correct and consistent specifications, it is hard to say whether a system design adequately addresses the requirements. To support some form of computer-aided analysis (that could help with determining its "appropriateness") a model needs to be formal to some extent, and thus the language in which it is expressed. Since the 1970s, many authors stated that natural language is not suited very well for expressing requirements (Teichroew 1972a, 1206) (Bell and Bixler 1976, 110)(Ross and Schoman Jr 1977, 9)(Zave 1979, 117)(Dubois et al. 1986, 1436) or specifications (Parnas 1972, 330). So artificial languages were proposed. An artificial language is formal by allowing only expressions that adhere to the rules of its grammar. Thus a subset of natural language is formed that is interpretable by machines. This facilitates checking e.g. for inconsistencies, errors, or undefined terms. The challenge to find what is the subset that is most appropriate for requirements engineering already has been taken multiple times.

To bridge the gap between customer needs and implementable specifications, many languages to support requirements engineering (from here on called requirements languages) were proposed, for example PSL/PSA (Teichroew 1972b), RSL (Bell and Bixler 1976), SADT (Ross and Schoman Jr 1977), PAISLey (Zave 1980), RML (Greenspan, Mylopoulos, and Borgida 1982), KAOS (Dardenne, Lamsweerde, and Fickas 1993), i*(Yu 1997), ADORA (Glinz, Berner, and Joos 2002), and URN (ITU 2008a). All have in common that they try to establish a structured way for the expression of requirements. Their difference is the kind of knowledge that can be modeled, whether the language is textual or graphical, whether it is accompanied by a constraint language, whether the abstract syntax of the language was specified with a meta-model, or whether it has formal semantics. For example, KAOS and URN are both graphical modeling languages, but KAOS is also accompanied by a "typed temporal first-order logic equipped with real-time temporal constructs" (Dardenne, Lamsweerde, and Fickas 1993, 13).

In spite of the wealth of requirements languages the majority of requirements specifications over the past 20 years are written in natural language (Hsia, Davis, and Kung 1993, p. 75)(Parnas and Madey 1995, p. 41)(Denger, Berry, and Kamsties 2003, p. 80)(Mich, Franch, and Inverardi 2004, p. 48)(Glinz 2010, p. 385)(Sikora, Tenbergen, and Pohl 2012, p. 57). While natural language is predominantly used, it cannot be seen as a a satisfactory requirements specification technique. Research is still working on defects in natural language requirements specifications (e.g. (Ott 2012)). Lack of industry adoption is also traced back to "methodical uncertainties" (Sikora, Tenbergen, and Pohl 2012, p. 61) regarding the capability of models to be legally binding agreements, the differentiation between requirements and design models, and how safety standards can be satisfied. In addition to these factors, we believe that existing requirements modeling languages do not have enough support for domain understanding and requirements elicitation. Before discussing the limited focus of these modeling languages, we first describe a definition of the early requirements phase, and knowledge to be captured in the early requirements model.

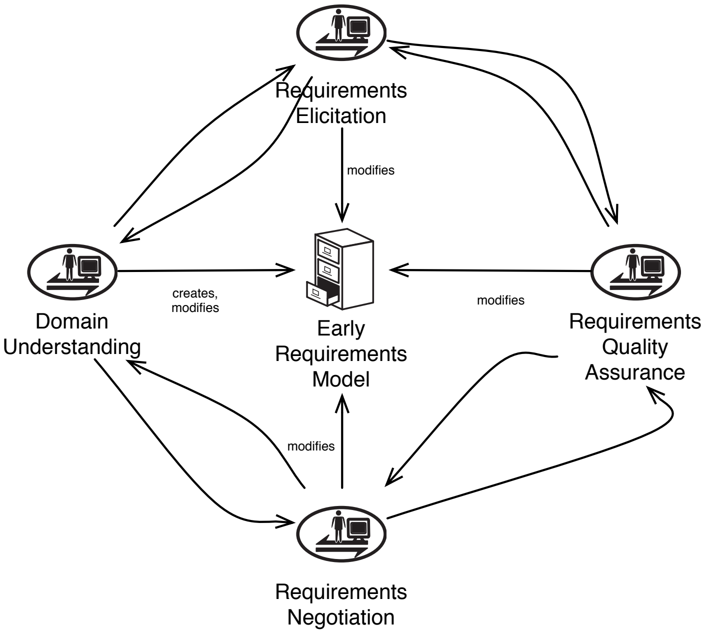


Figure 1.2.: Early requirements phase. Inspired by (Lamsweerde 2009)

Early requirements engineering consists of four activities (Fig. 1.2). These activities contribute to an artifact called the *early requirements model*. The sub-activities are tightly interweaved and are usually not occurring in a strict sequential order. Every activity can touch any element of the model, but usually a certain activity focuses on certain knowledge. During *domain understanding* the focus is on capturing domain-related knowledge. This includes identifying stakeholders (including the users) and asking them about their objectives with respect to the system. It includes understanding the processes in which the system will be embedded in. And it identifies existing products that are predecessors, might be similar to the new system, or competitng systems.
Requirements elicitation derives requirements from the knowledge gathered during the domain understanding phase. Some stakeholder requests might be directly translateable into requirements. Descriptions of how the system shall operate can be translated into use cases, from which functional and quality requirements can be derived. Use cases establish the system boundary. Actors can only interact with the system via its interfaces. Product management can request certain features directly. Stakeholder's goals can be analyzed with regards to how they can be realized by features. Each product feature can be broken down into a set of functional and quality requirements. For known dangers, experts can be asked for mitigating requirements.

Requirements quality assurance uses heuristics to enhance the completeness and consistency of the model. For example, a goal that cannot be satisfied by a feature could be discovered, or a feature that was not detailed into requirements. Some of these issues may be fixed by the analyst, but some might require validation by stakeholders.

Requirements negotiation prioritizes and ranks requirements and gives forecasts which requirements might be cost drivers. Preliminary road mapping, e.g. which features to plan for which version of the product, also takes place during this phase. If there are conflicting interests, they need to be settled by the stakeholders. For example, if two goals conflict, it has to be decided if one goal cannot be achieved by the system, or if different variants of a system could be feasible.

Several authors have made propositions what kind of knowledge should be expressed in such a model. Jacobson proposes to translate user requirements into use cases (Jacobson 1987). Kang et al. (Kang et al. 1990) propose requirements elicitation based on features. Dardenne et al. (Dardenne, Lamsweerde, and Fickas 1993) and Yu (Yu 1997) argue that goals should be modeled. Mylopoulos et al. discuss soft goals(Mylopoulos, Chung, and Nixon 1992). Lutz includes “hazard analysis early in the requirements analysis” (Lutz 1993, 131). She also argues that larger parts of the systems environment should be modeled, especially for embedded systems. Jackson and Zave et al. argue that domain knowledge should be distinguished from requirements, to enable showing that the requirements are consistent with knowledge about the domain (Jackson and Zave 1993; Gunter et al. 2000). Leveson argues that safety and security should be “built into the design from the beginning” (Leveson 2000, 16). In other words, a language integrating all these kinds of knowledge should provide an intentional view on the requirements, a product-oriented view, a behavioral view , and a protectional view. The intentional view models how stakeholder’s goals are broken down to requirements. The product-oriented view models which features system has and which features it has in common with other systems. The behavioral view models how and by whom the system will be used. The protectional view models which requirements provide protection from danger.

Current modeling languages either consider requirements as given and are merely interested in their connection to design, or, if explicitly dedicated to requirements modeling, prefer one view over the others.

UML (OMG 2011c), for example, has no abstraction to model requirements, it only supports modeling of use cases. SysML (OMG 2012), extends UML but is also focused on design. It can document requirements and trace them to design artifacts, but cannot differentiate between functional and quality requirements or provide any of the aforementioned views. Another language that is focused rather on specification than on requirements is Z (Spivey 1998). It allows to specify exactly in a mathematical style what a system is intended to to, but it does not allow to express why, and who has a stake in which part of the system. URN and KAOS focus on the intentional and behavioral view. FODA is dedicated to the product-oriented view.

In this dissertation, we propose the Unified Requirements Modeling Language (URML™) that integrates the intentional, product-oriented view, behavioral, and protectional views in a single model. Valuing one view over the other hinders the early requirements phase, as we would suppress readily available knowledge in favor of focusing on one particular view. By integrating the different views, URML provides a holistic perspective on requirements knowledge. URML's focus is on requirements elicitation in the sense of do Prado Leite: "for understanding, finding and gathering information"(Leite 1988). URML supports the views with a formal meta-model that integrates goals, features, dangers, use cases, stakeholders, and requirements concepts. That meta-model does not only provide a schema for repositories storing requirements models. It also is a model of the communication artifacts exchanged. By defining what a requirement is, what types of requirement exist, and what kinds of knowledge motivate requirements, it shapes the communication between analyst and the domain expert. With a given source meta-class, a valid instance of it and a given target meta-class, the meta-model can be traversed. From the relationships and entities encountered on the path, a question can be generated that the analyst can then pose to a stakeholder. Starting from any given model, one would generate a catalog of questions. Answers to a particular question can be used to populate the model - and generate more questions. To leverage the potential of a graphical modeling language for high usability, the notation of URML follows Moody's principles of graphical notation design (Moody 2009).

The dissertation is organized as follows. Chapters 2 and 3 gives a detailed account of the requirements knowledge and discusses related work with a focus on URN and KAOS, two languages from which URML was inspired. Chapter 4 presents the Unified Requirements Modeling Language (URML). A reference implementation is documented in chapter 6. In chapter 7, example models to evaluate the applicability of URML are created, and an experiment to evaluate the notations of URML, KAOS and URN is described. The dissertation is concluded in chapter 8. A detailed specification of URML can be found in appendix A

Typographical conventions used throughout this document *Italic* font is used to highlight key concepts. Whenever we refer to elements of a model in the text, we use `Typewriter` font.

2. Background

This chapter is dedicated to providing background knowledge and terminology needed for the remainder of this dissertation. Section 2.1 discusses the terms model and requirements model. Section 2.2 discusses requirements-specific concepts and provides the rationale why they should be modeled during requirements elicitation. Section 2.3 the structure of a modeling language is outlined and how meta-models support the definition of modeling languages. In section 2.4, works with focus on the design and evaluation of graphical notations are discussed. Section 2.5 is about use cases of tools supporting a requirements modeling language.

2.1. Model and Requirements Model

The central aspect of many definitions of the term *model* is its use of *abstraction*. Abstraction is defined by the Systems and Software Engineering Vocabulary (ISO 2010) as "a view of an object that focuses on the information relevant to a particular purpose and ignores the remainder of the information". Abstraction is also defined as the process of creating such a view. In this dissertation, we stick to the first definition of abstraction, if not mentioned otherwise. The relevancy of an abstraction is to be critically evaluated, as the leaving away of *relevant* details leads to vagueness. According to Dijkstra, "the purpose of abstracting is [...] to create a new semantic level in which one can be absolutely precise" (Dijkstra 1972).

Apart from the abstraction aspect, several definitions of the term model have slight differences in focus[1]. One difference is *what a model is about*. It can be about a system (Yourdon 1988; Davis 1993; Bruegge and Dutoit 2009; ISO 2010; OMG 2015c) or "target system" (Lamsweerde 2009), a "software component" (*Guide to the Software Engineering Body of Knowledge, Version 3.0* 2014), the "real world" (Jackson 1983) or "an existing reality" (Glinz 2014) or a part of the real world (ISO 2010) ("a real world process, device or concept"), concepts in general (Martin and Odell 1998), or a "reality to be created" (Glinz 2014). Glinz also provides more general definition, that uses "existing entity"/"entity to be created" instead of "existing reality"/"reality to be created" (Glinz 2014). Entity is defined in that context as "any part of reality or any other conceivable set of elements or phenomena, including other models". An earlier definition of the term entity defines it as "an element or set of elements that may stand for any conceivable item".

Another difference is the *purpose of a model*. Jackson refers to Ackoff to make a distinction between analytic and analogical models (Jackson 2001). Analytic models describe

[1] The definitions referred to in this section can be found in the appendix, Section A.11.

what is of interest in the real world. Analogical models create "another reality" that shares some similarity with a part of the real world (e.g. a flight simulator). Glinz makes a distinction between models that represent something existing and models that represent something to be created (Glinz 2014). Davis stresses the information providing aspect of a model: a model is only a model of a system if it can be "used to answer a well-defined set of questions about S to a tolerance adequate for a stated purpose" (Davis 1993). Bruegge and Dutoit point out that a model simplifies the reasoning about a system (Bruegge and Dutoit 2009).

The last difference that we have observed is regarding *what a model is*: In many definitions, a model is an expression, representation or description of something (Jackson 1983; Martin and Odell 1998; Lamsweerde 2001; ISO 2010; Glinz 2014; OMG 2015c), some definitions do not explicitly classify the form of a model (Yourdon 1988; Davis 1990; Bruegge and Dutoit 2009; *Guide to the Software Engineering Body of Knowledge, Version 3.0* 2014), and some make particular statements about what a model is: "abstraction device" (Martin and Odell 1998), the software itself (Jackson 1983), something a software works on (ISO 2010), "another reality" (Jackson 2001), or "an interpretation of a theory" (ISO 2010).

In this dissertation, we are concerned with a specific type of model, the *requirements model*. Requirements models may be analytical or analogical, or mix analytical and analogical representation. To support analogical models, executable behavior descriptions to simulate the desired behavior are required. These descriptions are either part of the requirements model or can be derived from it. The simulated behavior can be visualized to obtain feedback from stakeholders (Lamsweerde 2009). An example of an analogical requirements model is a model expressed in the PAISLey language (Zave 1980; Zave and Yeh 1981). Another example is the CREWS-SAVRE tool, which generates scenarios from use cases to perform scenario walkthroughs for the acquisition and validation of requirements (Maiden 1998). More tools for requirements simulation were surveyed by Schmid et al (Schmid et al. 2000). In this dissertation, we will focus on the analytical aspect of requirements models.

Glinz defines requirements model as a "model that has been created with the purpose of specifying requirements" (Glinz 2014). Together with Glinz' extended definition of model, this expands to "an abstract representation of an existing entity or an entity to be created[,] that has been created with the purpose of specifying requirements".

This definition is too narrow in terms of modality of any entity being modeled. It only allows present and future tense, or descriptive and prescriptive statements in van Lamsweerde's terms (Lamsweerde 2009). Certainly, a requirements model must describe parts of existing reality, as the requirements in the model shall be abstractions of the existing needs of the stakeholders. The stakeholders, existing persons or organizations, should also be represented by abstractions in the model. Furthermore, the environment of the system to be developed must be represented, which is also a part of existing reality. Any desired property or quality of a system to be developed that is represented in a requirements model can be considered as a reality to be created. But a requirements model must also represent phenomena that might but will not exist, and phenomena that are guaranteed to never exist.

For example, when a new product is developed to replace an older product and to improve an existing business process, the requirements model may contain an abstract representation of the functionality of the old product and of the existing business process, as well as the abstract representation of the desired functionality of the new product, and of the changed business process. The same model can also contain alternative representations of parts of the new product, because alternative sets of features were discussed to optimally support the new business process, and some of them were discarded because they were too expensive to realize. If the improvement of the aforementioned business process entails the absence of previously existing security issues, the requirements model will also describe phenomena that must not exist, along with features of the system to ensure that.

The definition above is ambiguous regarding what a requirements model abstractly represents. The singular entity in Glinz' definition gives the impression that a requirements model is an abstraction of a single thing. At the same time, entity is defined as "an element or set of elements". But a requirements model is about many things, including but not limited to the system to be developed, its environment, and its stakeholders. We therefore provide a refinement of the definition above, that addresses the two points of critique:

Definition *A requirements model is a set of abstract representations of entities that has been created with the purpose of specifying requirements. An entity represented in the model is a part of reality or any other conceivable set of elements or phenomena, including other models. The modality of the represented entities may differ.*

The question remains what exactly the requirements model should be an abstract representation of. In other words, what knowledge has to be in a requirements model to be considered adequate for a given stakeholder? Different authors have given different answers to that question. The next section on requirements terminology (Section 2.2) discusses a range of concepts that are candidates for being represented in a requirements model. As there are different schools of thought regarding what should be included in a requirements model, we would like to put Davis' pragmatic definition of the model term (Davis 1993) into focus, and adapt it regarding the requirements model:

Definition *RM is a requirements model of a system S, if RM can be used to answer a well-defined set of questions about the requirements for S to a tolerance adequate for a stated purpose.*

The second definition gives more freedom regarding the concrete ingredients of a requirements model. Any model can be a requirements model, as long as it can be used to answer questions of stakeholders regarding requirements. That the set of questions that can be answered is well-defined results from formally defining what can can be expressed in a model. Such a formal definition is the grammar of a modeling language. From this grammar, the kind of knowledge that potentially is encoded in a requirements model can be deduced, and thus the set of questions that can be answered.

The knowledge encoded in the model has to provide just enough detail so that any question can be answered “to a tolerance adequate for a stated purpose”. In this dissertation, the purpose of a requirements model is to support the communication of requirements analysts with stakeholders during requirements elicitation.

Goal URML supports the communication of requirements analysts with stakeholders.

2.2. Requirements Terminology

This section discusses the concepts to be integrated in our language. As most of these concepts do not only have one commonly accepted definition, we discuss several definitions, to motivate more than one aspect of the concept that is useful for requirements analysis. The definitions that we have chosen for URML are presented in chapter 4.

System The *system under discussion* (SUD) is the system that is being developed, and whose requirements are therefore being discussed. Usually that system is only put on diagrams to show where the boundary of the system lies, i.e. what is inside and what is outside of the system. In a requirements model for systems-of-systems it can become necessary to make the discussed systems explicit in order to make traceable which system has which requirements.
In structured analysis and design, the requirements modeling effort usually starts with a context diagram. It supports understanding the environment of the system, or the boundary between the “system and the rest of the world”. This is achieved by adding the system, terminators, the data used and produced by the system, and data stores shared between the system and the terminators on the diagram (Yourdon 1988). The system is represented by a circle in the center of the diagram.
For object-oriented analysis and design, Jacobson proposes to start the requirements modeling effort with the use case model. According to Jacobson et al., one of the purposes of a use case model is to delimit the system (Jacobson et al. 1992). In his use case diagrams, a rectangular shape represents the system. That shape separates the use cases from the actors on the diagram. In UML, the same rectangular shape is called system boundary, or subject of the use case (OMG 2013b).
Martin and Odell mix the ideas of context diagram and use case diagram to create a UML flavor of context diagrams, where the notation of use case is used to represent the system under discussion (Martin and Odell 1998).
The Unified Process provides extensions to UML that allow to stereotype packages representing the system under discussion on different levels of abstraction, ≪use case system≫, ≪analysis system≫, ≪design system≫, and ≪implementation system≫ (Jacobson, Booch, and Rumbaugh 1999).

Jackson criticizes that the term system may be used ambiguously by stakeholders, as it is common that the combination of the system and its environment is also called system (Jackson 2001). He suggests to call “what must eventually be built” the “machine”.

Requirement Required functions are usually modeled in the form of *functional requirements*. The term “functional requirement” appeared early in the requirements literature (Beckenstein 1972). The focus on functionality alone was criticized by Yeh and Zave (Yeh and Zave 1980).

Quality requirements should be modeled because the purposefulness of a system not only depends on the set of functions it has, but also the appropriateness of these functions (Chung and Leite 2009, p. 363). Several taxonomies have been proposed for quality requirements, most notably the international standard ISO/IEC 9126 (ISO 2001), recently superseded by the ISO 25000 family of standards (ISO 2005). Quality requirements are a specific category of non-functional requirements (Yeh and Zave 1980) (NFR). NFRs comprise all requirements that deal with other aspects than functionality (Chung and Leite 2009, p. 366). The second category of NFRs is called constraints. Constraints are requirements not targeted at the functionality or quality of the to-be-developed system, but at limiting “the solution space beyond what is necessary for meeting the given functional requirements and quality requirements” (Glinz 2014).

Berenbach et al. describe prioritization and ranking of requirements (Berenbach et al. 2009, 53).

Stakeholder The term *stakeholder* has been introduced to requirements engineering by Macaulay (Macaulay 1993). She defines stakeholders as “all those who have a stake in the change being considered, those who stand to gain from it, and those who stand to lose”. Kotonya and Sommerville define it as “people or organizations who will be affected by the system and who have a direct or indirect influence on the system requirements” (Kotonya and Sommerville 1998, p.10). Stakeholders should be modeled in order to understand who has an interest in the system (Sharp, Finkelstein, and Galal 1999). Alexander and Robertson define a project stakeholder as “someone who gains or loses something [...] as a result of that project”(Alexander and Robertson 2004, p.23). Failure to identify all relevant stakeholders may lead to the development of a system that is not wanted or lacks functionality for certain users. The importance of involving “relevant stakeholders” is also stressed by the “CMMI for Development” report (CMMI Product Team 2010). The most recent definition we are aware of is given by Glinz: “A person or organization that has a (direct or indirect) influence on a system’s requirements. Indirect influence also includes situations where a person or organization is impacted by the system” (Glinz 2014).

Users are an important class of stakeholders. They should be modeled in order to understand who will be using the system. Gause and Weinberg state that users should be included in the development process “before any design decisions are made, and the earlier the better” (Gause and Weinberg 1989, p. 78). Their notion of user is so broad that it contains customers (which might not be users) and losers (groups of people that intentionally may not use the product).

Users are regarded as instances of *actors* by Jacobson et al. Actors are anything that “interacts with the system” and “[....] needs to exchange information with the system”. This definition entails “other systems communicating with our system” (Jacobson et al. 1992). Yu introduced the *intentional actor* for discovering strategic dependencies between actors (Yu 1997). Following Yu's definition, actors have goals and depend on other actors to achieve these goals.

Yourdon uses the term *terminator* to model external entities with which the system under discussion interacts. Terminators include persons, groups of persons, but also larger organizations, or other systems (Yourdon 1988).

Dardenne et al. define an *agent* as "an object which is a processor for some actions; agents thus control state transitions. As opposed to the other kinds of objects (i.e., entities, relationships, and events), agents have choice on their behavior" (Dardenne, Lamsweerde, and Fickas 1993). This abstraction is based on Yue's definition of agent: an agent is "an active component of a system. It can initiate events." (Yue 1986) In (Lamsweerde 2001) van Lamsweerde interprets Yu's intentional actor as agent. In (Lamsweerde 2009) van Lamsweerde differentiates between Software-To-Be-Agents and Environment Agents. Software-To-Be-Agents are components of the system under discussion. Environment agents are similar to Jacobson's actors, they are entities with behavior external to the system.

Goal A *goal* is an objective that a stakeholder wants to achieve. A goal affects the system under discussion, either because it influences the system, or because the system itself is responsible for achieving the objective. Goals are either operationalized to functional requirements or translated to non-functional requirements or constraints on the operation of the system (Mylopoulos, Chung, and Nixon 1992, p.3). Yu's colloquial approach to delineate goals from requirements is that goals address the "whys" of a system instead of the "whats"[2](Yu 1993, p.34)(Yu 1997, p.226).
Systems viewed as agents or actors can have goals themselves. As the "goals of an agent may not coincide with his capability", a "need for cooperation and communication arises"(Yue 1987). That need for cooperation is modeled in Yu's strategic dependency diagrams (Yu 1997). Such diagrams can show how an actor A depends on another actor B to perform some task in order to get A's goal satisfied.
Goals range from business objectives (p.226) to user goals. They provide the rationale for requirements decisions (Mylopoulos, Chung, and Nixon 1992, p.2) or are justifications for requirements ("understanding the "whys" that underlie system requirements" (Yu 1997, p.226)). Goals often do not have one possible operationalization in terms of requirements. They require the discussion of these alternatives before design time. Goals can also be used late in the requirements process for completeness checks on the set of known requirements. Yue proposes goals for checking the completeness of requirements specifications expressed in terms of a constraint language (Yue 1987).

Resulting from the need to model goals, an "early phase" is introduced to the requirements process that takes place before requirements can be specified in implementable form. Yu's "early phase" (Yu 1997, p.226) is called "requirements acquisition" by Dardenne et al (Dardenne, Fickas, and Lamsweerde 1991, p.14)(Dardenne, Lamsweerde, and

[2] The two terms are quoted as originally used by Yu. The "whats" of a system are the implementable requirements.

Fickas 1993, p.5).

To summarize, goals uncover the intentions behind requirements and help with reasoning about them. An overview on the modeling of goals is given by van Lamsweerde (Lamsweerde 2001).

Feature The American Heritage Dictionary of English Language (AH), defines a *feature* as a "prominent or distinctive part, quality, or characteristic" (AH 2011). This definition was adopted by Kang et al., who introduced the term to the software engineering domain: "A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems", describing "capabilities from the perspective of end-users" (Kang et al. 1990, p.3,23). Features are an effective means of communicating with stakeholders used by many stakeholders to intuitively describe product characteristics (Kang et al. 1998, p.144)(Kang, Lee, and Donohoe 2002, p.58)(Lee, Kang, and Lee 2002, p.64). Because a feature can characterize the difference between a product and another, features enable product line modeling. They enable the study of the variations and commonalities between the products of a product line (Kang et al. 1990, p.1). To enable the planning of reuse (Lee, Kang, and Lee 2002, p. 64), Kang et al. advocate to distinguish between domain engineering and application engineering (Kang et al. 1990, p.1): during domain engineering, the application domain is studied and the knowledge gathered is captured in the form of a feature model. During requirements analysis for a concrete product, the domain model is used to select the features of the product under discussion. This means before any development process is started, a domain model has to exist. Kang et al. believe that features should be first class objects for the whole development process. Kang proposes a modeling approach called "Feature-Oriented Domain Analysis (FODA)", which is explicitly dedicated to requirements engineering. However, the relationship between the concepts requirement and feature is left unclear. Later, Kang et al. state that features are "functional abstractions" (Kang et al. 1998, p.144), Lee at al. add "... of requirements" (Lee, Kang, and Lee 2002, p.64). We argue that this view on requirements is too general on the one side and too narrow on the other. It is too general, as requirements analysts want to retain the information which functions are associated with a specific feature, to deal with conflicting views among stakeholders what exactly a specific feature entails. It is too narrow, as the focus on functions neglects that features implicitly contain quality requirements. The definition of Simos et al. (Simos et al. 1996) is therefore more suitable as it explicitly includes system behavior: "A feature is used to express a differentiating behavior or characteristic associated with a system artifact that is significant to some domain practitioner."

Classen et al.'s definition relates a feature to the set of requirements it satisfies (Classen, Heymans, and Schobbens 2008, p.21), but does not restrict these requirements only to functional requirements: "A feature is a triplet, f = (R,W,S), where R represents the requirements the feature satisfies, W the assumptions the feature takes about its environment and S its specification."

Danger Even with the best intentions of the designers and developers, systems may have safety issues, which are called *hazards*. Leveson defines a hazard as "A system state or set of conditions that, together with a particular set of worst-case environmental conditions, will lead to an accident (loss)" (Leveson 2011).
Requirements elicitation has to put extended focus on uncovering potential hazards. Of 96 discovered safety-related faults on the software of the Galileo spacecraft, 76 could be traced back to "errors in recognizing requirements" (Lutz 1992, 1993) (See appendix sectionA.12 for a visualization). When faults are found in an already operating system, consequences may be catastrophic or very expensive to fix. Lutz and Mikulski report on an example where eleven new requirements were identified when investigating 44 "safety-critical operational anomalies" (Lutz and Mikulski 2003, p. 157). Lutz and Woodhouse describe a hazard analysis technique that uncovered new failure modes (Lutz and Woodhouse 1996, p. 45) and thus modified existing requirements.
Hazard analysis often is required for systems where erroneous system behavior may lead to physical harm, e.g. in the military, where contracts may require compliance with MIL-STD-882 (DoD 2012), or for medical systems (see ISO 15189).

When a system becomes the starting point of a product line, it is beneficial for the development of all subsequent systems to already have the knowledge about potential hazards identified earlier. If the hazard analysis results are reported in a different document than the design or requirements specifications, knowledge might get lost in subsequent projects. *Mitigations* show how a system minimizes the probability of a hazard or its severity. Mitigation of hazards is a quality characteristic of a system in use (ISO 2011, p. 3). Any mitigation can be understood better if the hazard that it mitigates is part of the requirements model. Even when a mitigation is not realized, for instance because the hazard's probability was estimated as being unlikely, it is beneficial to record existing knowledge regarding that mitigation, once changed circumstances call for a re-evaluation of that decision (Lutz and Mikulski 2003, p. 158).

Similar to safety concerns, *threats* are modeled for security concerns, where values like money, identity, confidential information are at risk. A threat is defined as "a state of the system or system environment which can lead to adverse effect in one or more given risk dimensions" (ISO 2010).

Sindre and Opdahl propose misuse-cases as a counterpart to use cases for elicitation of security requirements (Sindre and Opdahl 2000). CORAS suggests a UML profile to model threats, including a diagram type inspired by misuse cases (Braber et al. 2007; OMG 2008). Elahi et al extend the i* framework to support graphical modeling of vulnerabilities and threats during requirements elicitation (Elahi, Yu, and Zannone 2009). Matulevičius et al. have extended the Secure Tropos framework for the same purpose (Matulevičius et al. 2008).

Potts introduces *obstacles* as the negative counterpart to goals (Potts 1995). If an obstacle is present, the goal might not be achieved. The abstraction of an obstacle has been incorporated in the KAOS meta-model. The duality of obstacles and goals is described by van Lamsweerde and Letier (Lamsweerde and Letier 1998). For example, if an obstacle obstructs a safety goal, it is a hazard. If an obstacle obstructs a security goal, it is a threat. Van Lamsweerde describes an extension to the KAOS meta-model that includes a `SecurityGoal` meta-class that is further specialized to the subclasses Confidentiality, Integrity, Availability, Privacy, Authentication and Non-repudiation (Lamsweerde 2004). In van Lamsweerde's textbook (Lamsweerde 2009), that meta-class is reduced to an enumeration literal of the Goal meta-class.

IEEE 1012 proposes a four-level integrity scheme for software components. On level 4, malfunctioning components have "grave consequences", i.e. lead to "loss of life, loss of system, economic or social loss", where "no mitigation is possible". On level 3, malfunctioning components have "serious consequences", i.e. lead to "permanent injury, major system degradation, economic or social impact", and "partial to complete mitigation is possible". For software components with integrity level 3 or 4, hazard and security analysis tasks are to be performed in all activities of the system development and operation processes (IEEE 2005)[3].

Process The *workflows* of the organization where the system under discussion will be used must be studied to understand how the organization imposes functional, quality and interface requirements on the system. Robertson and Robertson advocate "quick and dirty process modeling [...] to gain understanding and consensus about the current work" (Robertson and Robertson 2012).

The Business Process Model and Notation (BPMN) defines *business process* as a "defined set of business activities that represent the steps required to achieve a business objective. It includes the flow and use of information and resources" (OMG 2011a). In BPMN, workflows are called "private business process".

The *use cases* of the system under discussion must be modeled and added to the requirements model in order to understand how the system will be used. Jacobson et al. define a use case as "a behaviorally related sequence of transactions in a dialogue with the system" (Jacobson et al. 1992). Dano et al. as well as Rolland and Ben Achour propose a use case driven requirements acquisition (Dano, Briand, and Barbier 1997; Rolland and Ben Achour 1998).

Potts defines *scenarios* as "narrative descriptions of interactions between users and proposed systems" (Potts 1995). According to Bruegge and Dutoit, a scenario is an instance of a use case (Bruegge and Dutoit 2009). That view is supported by Cockburn (Cockburn 2000): a "use case gathers [...] different scenarios together".

Object A *class* is an abstraction that describes the commonality of a set of objects in terms of structure and behavior (Jacobson et al. 1992; Bruegge and Dutoit 2009). Classes support inheritance. A sub-class inherits the attributes and operations of the super-class. An *object* is an instance of a class. This means it has a concrete identity and all of its properties have a value.

Jacobson has introduced three special kinds of objects to be used during analysis: *entity, boundary, and control objects* (Jacobson et al. 1992). *Interface objects* allow for modeling with which parts of the interface actors interact, and through which such parts

[3] IEEE 1012 follows the process model of ISO/IEC 12207:1995.

they participate in use cases. *Entity objects* “model the information that the system will handle over a longer period of time”, typically they live longer than the duration of a single use case. *Control objects* are needed when the use case is so complex that the coordination between interface and entity objects has to be encapsulated in one object. Typically there is a one-to-one relationship between control object and use case.

2.3. Structure of a Modeling Language

According to Kleppe (Kleppe 2008), a language specification consists of three mandatory and three optional parts. The mandatory parts are “an abstract syntax model (ASM)”, “one or more concrete syntax models (CSMs)”, and “for each concrete syntax model, a [...] syntax mapping”. The optional parts are a definition of the meaning of the language expressions, “including a model of the semantic domain”, “the required language interfaces”, and the “offered language interfaces”. Except the language interfaces, these parts are also mentioned in Selic’s graphical model of graphical modeling language (Selic 2011) in Figure 2.1. In this dissertation, we use Selic’s terms. The abstract syntax contains the concepts of the language, and rules how these can be composed. A modeling language has at least one concrete syntax, which describes how expressions in a language can be uttered. It entails notational elements and rules how these can be composed. According to Selic’s model, there is not necessarily a one-to-one mapping between notational elements and language concepts. The structure of the concrete syntax is possibly different from the structure of the abstract syntax.

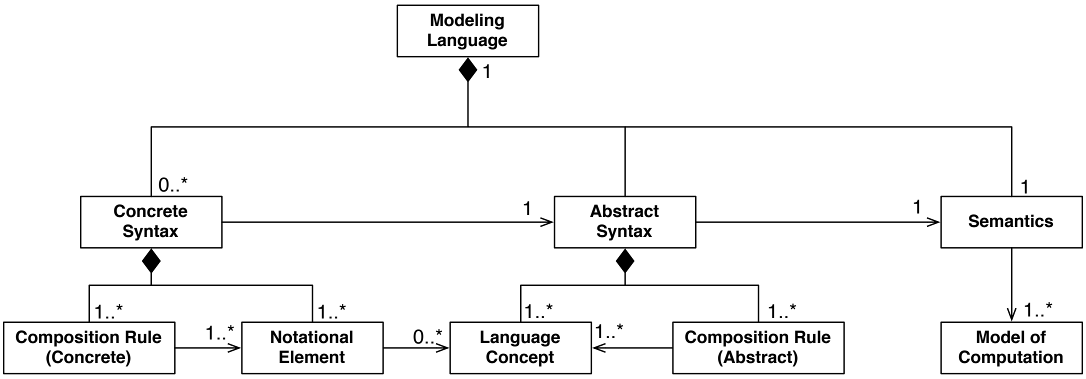


Figure 2.1.: Structure of a modeling language. Adapted from (Selic 2011).

In this dissertation, we focus on the abstract syntax of URML and one particular concrete syntax, which is graphical. Both are described in Chapter 4. This section therefore describes how abstract and concrete syntax of a graphical modeling language can be defined. Many standards published by the Object Management Group (OMG) are based upon the Meta Object Facility (MOF). MOF is a meta-language with which the abstract syntax models of modeling languages can be defined. Figure 2.2 gives an example of how this works. The figure shows excerpts of three models that build a hierarchy, each model on a different layer of abstraction. The topmost layer, showing an excerpt of MOF, is instantiated on the layer below, showing an excerpt of UML, which in turn is instantiated on the lowermost layer, which shows an excerpt of a hypothetical user model.

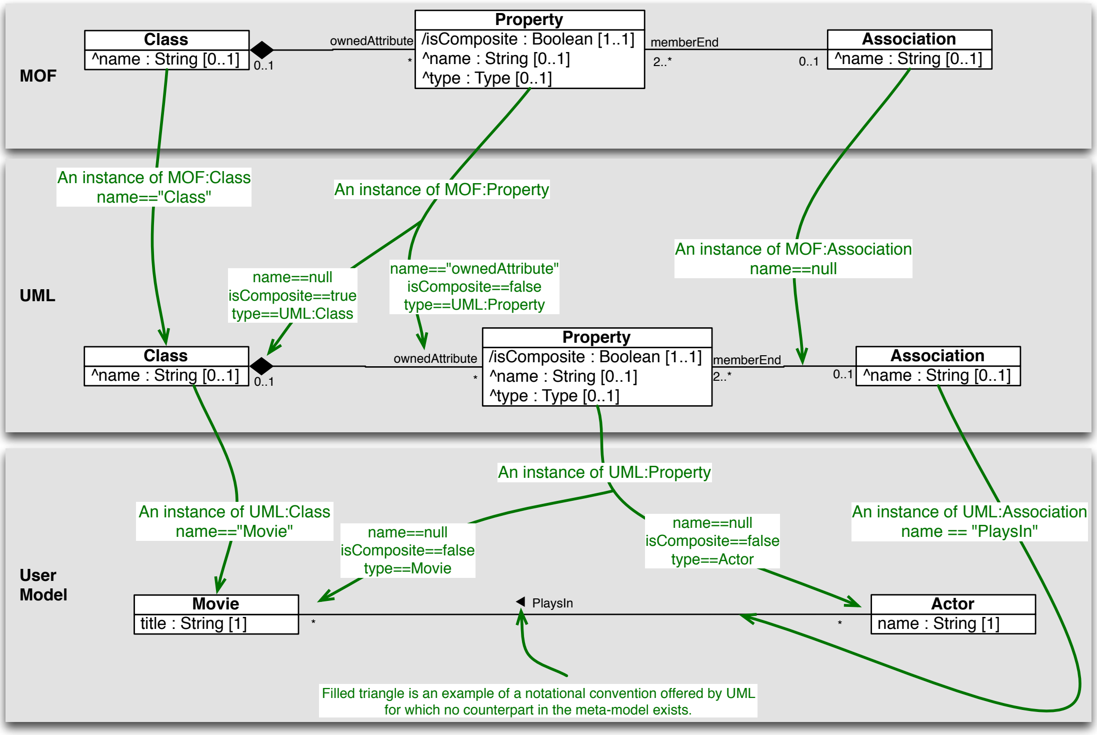


Figure 2.2.: UML model hierarchy. Inspired by similar figure in (OMG 2011b).

The relationship between MOF and UML is special as MOF itself reuses and extends the Core package of UML (OMG 2015b). In Figure 2.3 another example is presented, showing a part of how the Interaction Flow Modeling Language (IFML) is defined on the basis of MOF (OMG 2015a). The basic MOF abstractions `Class`, `Property`, and `Association` are instantiated on the IFML layer to model a central aspect of IFML: An `InteractionFlow` maps a source `InteractionFlowElement` to a target `InteractionFlowElement`. Concrete subtypes that can occur on diagrams are `ViewContainer` and `ViewComponent`. `InteractionFlow` is also abstract. A concrete subtype shown in the figure is for example `NavigationFlow`. In the user model layer, a `ViewContainer`, two `ViewComponents`, and a `NavigationFlow` are instantiating elements of the IFML layer.

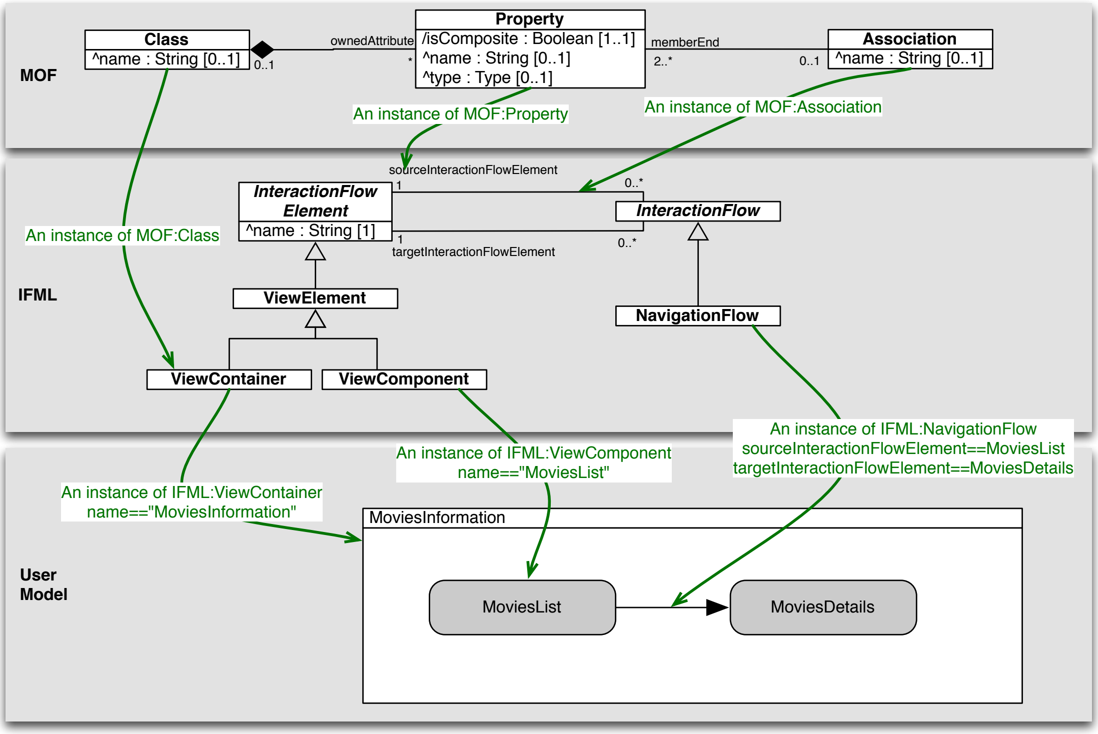


Figure 2.3.: IFML model hierarchy. IFML metamodel excerpt adapted from (OMG 2015a).

A meta-language comparable to MOF is GOPRR (Kelly 1997). Kelly and Tolvanen claim that it was specifically designed for the description of modeling languages (Kelly and Tolvanen 2007). The acronym stands for the main abstractions of the language: Graphs, Objects, Properties, Relationships, and Roles. Figure 2.4 shows an excerpt of the GOPRR model on the left-hand side. The notation used is one specifically invented for describing the GOPRR model to avoid the recursion that would have occurred when GOPRR would be used to define itself (as is the case with MOF). Kelly states that the model is based on object-oriented methods (Kelly 1997). The abstractions of GOPRR are described with their name, which is, in bold font, in the top compartment of the rectangle signifying the abstraction. The other two compartments hold instance (middle compartment) and class (lower compartment) variables[4]. On the right side, the GOPRR model is instantiated to model a dataflow diagrams.

[4] The types of the variables are also not present in the original version of the figure in (Kelly 1997), but can be found in the textual description of the model in the same work.

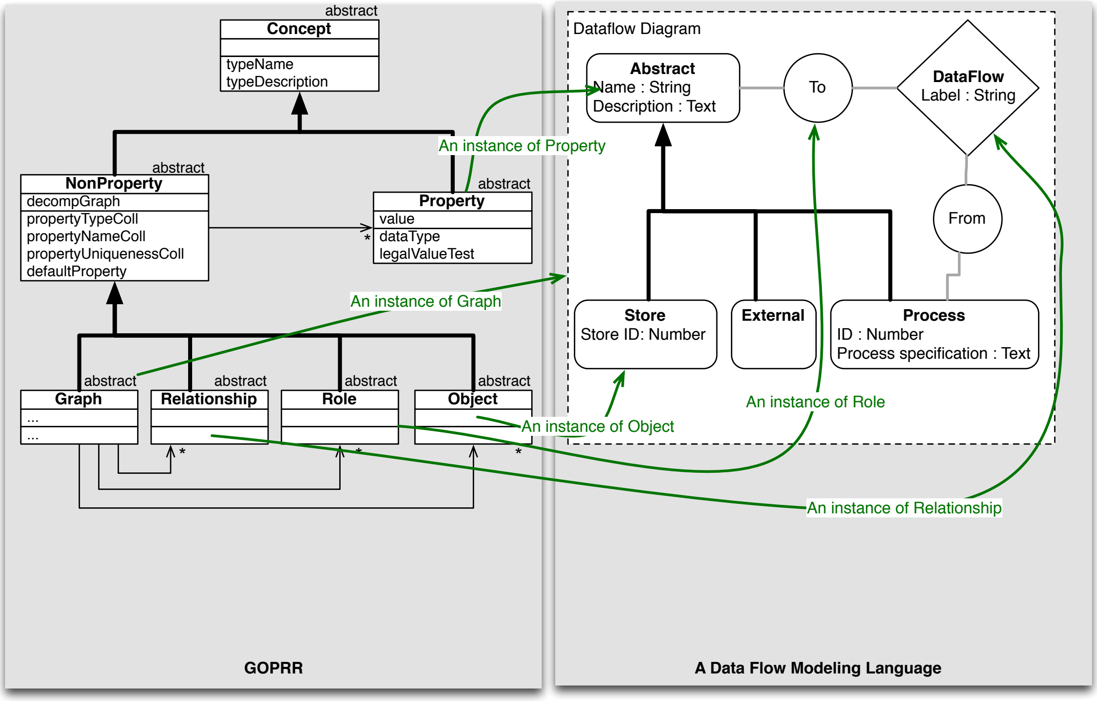


Figure 2.4.: GOPRR model hierarchy. Meta-meta model adapted from (Kelly 1997), meta-model from (Kelly and Tolvanen 2007).

For pragmatic considerations detailed in Chapter 6, MOF was chosen as the basis for the abstract syntax model of URML. Apart from the pragmatic considerations, basing URML on MOF enhances its comparability with URN (described in Section 3.2) and KAOS (described in Section 3.1): URN is also based on MOF as recommended in (ITU 2008b), and for KAOS it is stated that the basic principles of UML class diagrams are used to specify the KAOS meta-model in (Lamsweerde 2009).

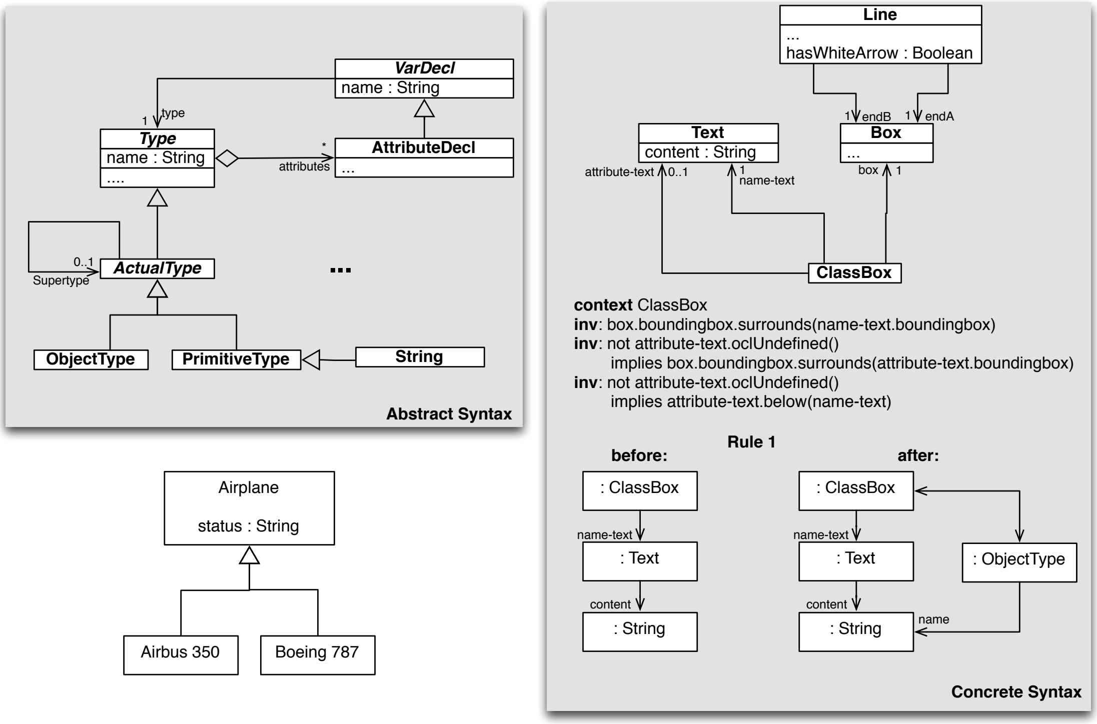
Requirement The abstract syntax model of URML shall be defined using MOF.

MOF itself offers two levels of compliance, called Essential MOF (EMOF) and Complete MOF (CMOF) (OMG 2015b). Basically both compliance levels allow for a simple form of UML class modeling. The MOF specification explicitly specifies which packages of the the UML are merged into the MOF meta-model, and EMOF and CMOF respectively specify which classes of these packages may be used for meta-modeling. The main difference of CMOF to EMOF is that CMOF allows some more meta-classes, to model dependencies between packages (ElementImport, PackageImport, and PackageMerge) and Constraints (Constraint and OpaqueExpression). As the necessity for modeling constraints was envisioned for the URML meta-model, full CMOF compliance is desired for URML.

Requirement URML's abstract syntax model shall be CMOF-compliant.

Regarding how to specify the concrete syntax of a modeling language, Selic notes that “in case of graphical languages, there is no satisfactory agreed on method for specifying a notation” (Selic 2011). The UML specifications have followed an informal approach, in which the notational elements are described textually. These descriptions are then accompanied by example figures. One problem with this approach is, that it leaves room for interpretation. The combinatorial variety of possible instantiations of a simple meta-model graph quickly gets out of hand, the more properties the participating meta-classes have. This is aggravated by the fact that UML often offers additional options for the representation. For example, compartments in class symbols may be suppressed, or whether a class is abstract or not may be indicated by italic font or an ≪abstract≫ keyword.

Kleppe describes the constituents of a concrete syntax in more detail (Kleppe 2008): A “complete concrete syntax description” consists of an *alphabet*, *scanning and parsing rules*, *abstraction rules*, and *binding rules*. The alphabet defines which notational elements do exist and how they can be composed. The scanning and parsing rules specify how combinations of notational elements can be interpreted as syntactical constructs. Abstraction rules specify “which parts of the concrete form do not have a direct counterpart in the abstract form”, and binding rules “where and how separate [...] concrete elements represent the same abstract element”. Kleppe gives an example in which the alphabet is specified with a MOF (or UML) model. It describes the structure of the notational elements and its associations describe a part of the scanning and parsing rules. Constraints expressed in Object Constraint Language (OCL) are added to the alphabet to specify those scanning and parsing rules that cannot be expressed by classes and associations. The abstraction and binding rules are specified with graph transformations. An excerpt of Kleppe’s example is reproduced in Figure 2.5.



```
context ClassBox
inv: box.boundingbox.surrounds(name-text.boundingbox)
inv: not attribute-text.oclUndefined()
       implies box.boundingbox.surrounds(attribute-text.boundingbox)
inv: not attribute-text.oclUndefined()
       implies attribute-text.below(name-text)
```

Figure 2.5.: Excerpts from abstract syntax, concrete syntax, and a user model of the Alan language. Adapted from (Kleppe 2008).

To summarize, Kleppe stipulates that the concrete syntax of a modeling language is described by a meta-model (with constraints) and graph transformation rules.

Fondement and Baar have proposed a technique (Fondement and Baar 2005; Baar 2006) in which the mapping between notational elements and abstract syntax classes is defined in a dedicated layer of "display managers". In Selic's model, shown in Figure 2.1, there is a tight coupling between NotationalElement and LanguageConcept realized by a number of associations. The display manager layer serves as an intermediate layer between the abstract and concrete syntax models. It thus decouples the two layers and organizes the mapping of abstract syntax meta-classes to notational element meta-classes. The organization is realized by constraints defined in the display manager classes. The example in Figure 2.6 shows an excerpt of a simplified meta-model for state charts on the left: A `Transition` may be triggered by an `Event`. On the right hand side is a structural definition of the representation of such `Transitions`: `TransitionRepresentation` is concerned with the routing of the line, it contains two `ArrowHeads`, one for the start and one for the end of the transition, and it optionally contains a `Text` for the label holding the `Event` name. The `Text` instance will only be needed, if a `Transition` has a triggering `Event`. This constraint is maintained in the `TransitionDM` class, specified using OCL.

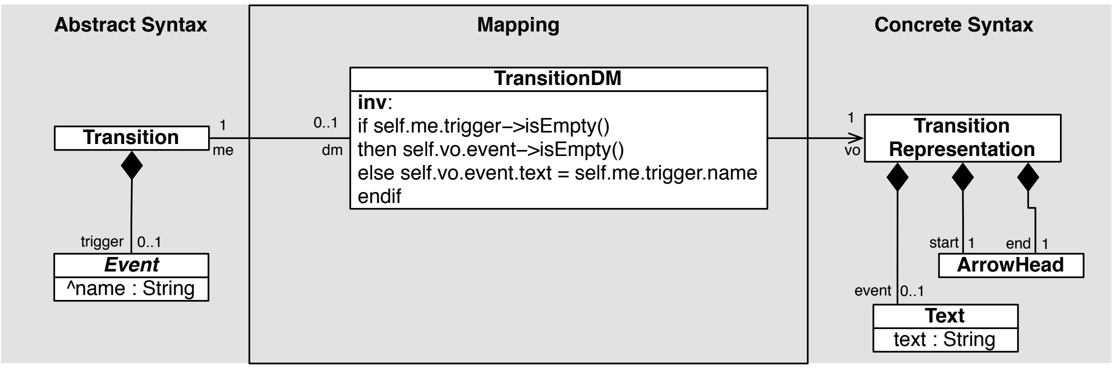


Figure 2.6.: Example outlining Fondement's and Baar's approach to concrete syntax specification. Adapted from (Fondement and Baar 2005).

Goal The concrete syntax specification of URML will follow the architecture defined by Fondement and Baar.

Convention For the abstract-concrete-syntax-mapping classes we will make use of the optional UML notation in which classes may have a compartment dedicated to constraints. The compartment will not be prefixed by the term "constraints:".

2.4. Principles for Graphical Notation Design

In "Semiology of Graphics", Bertin defines eight ***visual variables*** for encoding information on a two-dimensional plane (Bertin 1967)(Bertin and Berg 2011). The first two, the *planar variables*, are the *two dimensions of the plane*. The other six, the *retinal variables*, are *size*, *value*, *texture*, *color*, *orientation*, and *shape*. In Figure 2.7, examples are given how the retinal variables can modify the appearance of a graphical mark. Such a mark visualizes a "pertinent correspondence". According to Bertin, the information transcribed into a diagram is a "series of correspondences observed within a finite set of variational concepts", the components. The components are relating to an "invariable common ground", the invariant. For example, in a diagram showing a trend of a stock on a certain exchange, the invariant is the quotation in a certain currency as the closing price for cash payment on a certain exchange. It has two components, the actual price, and the day on which the price was noted. In this example, the two planar variables are enough to encode the two components. If the creator of the diagram wanted to add the trend of a second stock, a third component would need to be visualized: the identity of the stock. One retinal variable would be needed to make the two trend lines distinguishable, e.g. using a thicker line for the second stock (size). If the diagrammer wanted to add a fourth component, e.g. a second stock exchange, to compare how the two stocks were noted on two different exchanges, a second retinal variable would be used, e.g. texture to distinguish the trend lines of the two stocks on exchange A from the trendlines of the two stocks on exchange B (example adapted from (Bertin and Berg 2011)). In short, the number of visual variables used should be greater or equal than the number of components of the graphic. In the case where the number of visual variables is greater than the number of components, there is a redundant use of several variables for one component that can increase the perceptible difference between two marks.

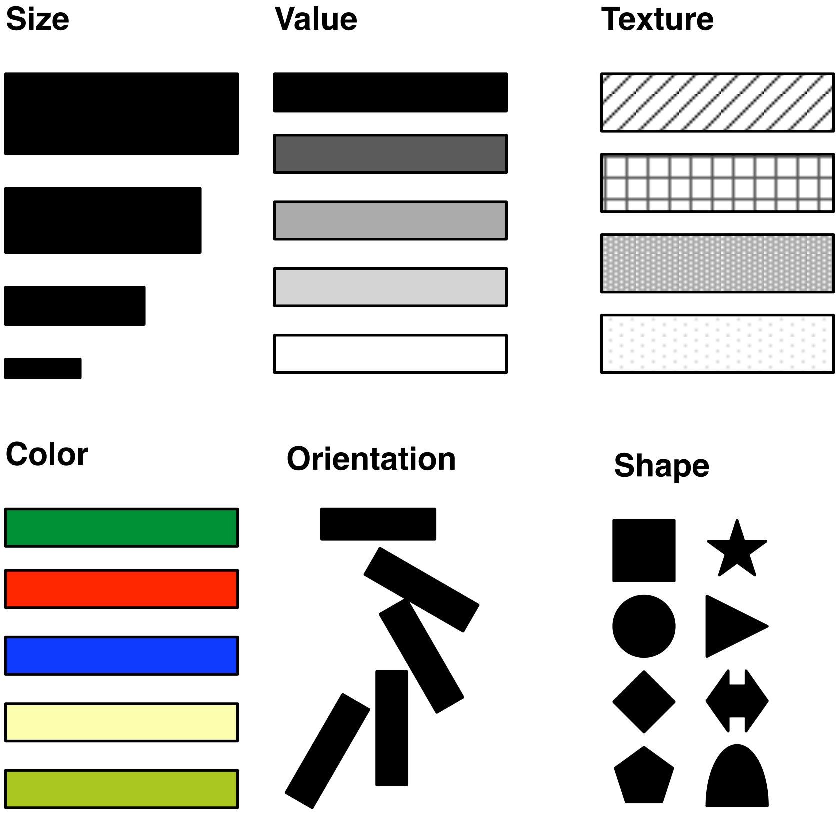


Figure 2.7.: The six retinal variables and examples. Adapted from (Bertin and Berg 2011).

In order to describe how the visual marks can be organized on the plane, Bertin differentiates between ***implantation***, ***level of organization***, and ***imposition***. ***Implantation*** classifies the graphic according to three classes of representation: the marks can either be represented as points, lines, or areas. In a single graphic the same concept should not be represented using different classes. The implantation influences the representation of quantities (i.e. when the size visual variable is applied, a point becomes larger, a line thicker).

The ***level of organization*** of a visual variable characterizes what it can be used for: *Associative* variables enable an immediate grouping of all the correspondences the variables differentiate. If the reader is unable to immediately spot all the marks varied by the variable, the variable is *dissociative*, an effect that happens when the variable varies the visibility of the mark (e.g. size). *Selective* variables enable the immediate isolation of all correspondences of the same category. If the reader is unable to spot all marks belonging to the same category, the applied variable is not selective. *Ordered* variables enable the immediate recognition of a universal sequence of steps in the marks. If the ordering of the marks is not apparent to the reader, such a variable is not ordered. *Quantitative* variables allow for an immediate quantification of the distance between the steps of an ordered variable. If the reader is unable to immediately deduce a value from a sign, given another sign is representing value one, it is not quantitative.

Imposition classifies a graphic into four types: *diagrams*, *networks*, *maps*, and *symbols*. For *diagrams*, for each value of a given component there is a value in the other components. In *networks*, only correspondences exist between the values of a single component. *Maps* are similar to networks in that the correspondences only exist among the values of one component. In addition, there always is a component locating the geographic entities on the plane. For *symbols*, there is no correspondence that is visible on the plane. The correspondence is exterior to the graphic, for example, if the symbol has a culturally defined meaning accepted among a group of people. Some impositions can be further subclassed by an *imposition type*. Diagrams can have the imposition types *rectilinear*, *circular*, *orthogonal*, and *polar*. Networks can have the imposition types *arrangement*, *rectilinear*, *circular*, and *orthogonal*. Maps and symbols can only have the imposition type *arrangement*.

To put Bertin in relation to the modeling languages we discuss in this dissertation, what we call a diagram, would be called a network by Bertin.[5] The networks based on modeling languages usually have two components: one is the entity type and the second the relationship type. Depending on the semantics of the relationships used, these networks are either arbitrarily, rectilinearly, or circularly arranged. An orthogonal organization of networks is found in many CASE tools that support a relationship matrix. The visual variables used on these networks are usually shape (e.g. in UML, a rectangular shape is representing `Class`) and texture (e.g. `Include` relationships are represented with lines that have a dashed pattern).

Moody's "Physics of Notations" (Moody 2009) is a framework for graphical notation design that contains nine principles, to lead to user-friendly graphical notations. The nine principles are Semiotic Clarity, Perceptual Discriminability, Semantic Transparency, Complexity Management, Cognitive Integration, Visual Expressiveness, Dual Coding, Graphic Economy, and Cognitive Fit. The framework constitutes an amalgamation and application of published and evaluated work on semiotics and cognitive psychology to notation design. It is based on the work of Bertin (Bertin 1967) and the Cognitive Dimensions framework (Green and Petre 1996). Sousa et al. claim that Moody's principles are also based on earlier work of Constantine and Henderson-Sellers (Sousa et al. 2012). In the following, we will give a short explanation of the principles.

Semiotic clarity[6] prescribes an ideal one-to-one correspondence of language con-

[5] In this paragraph we stick to Bertin's terms, but will fall back to the notion "diagram" in other parts of this dissertation.

[6] The present paragraph is a summary of the according sections of the aforementioned paper of Moody. It is largely phrased in our own words. Still, for reasons of readability, we renounced using quotation marks wherever the original text is cited in verbatim.

structs[7] to symbols. *Symbol redundancy*, *symbol overload*, and *symbol excess* are undesirable properties of a language with respect to semiotic clarity. Symbol redundancy means that constructs exist that have more than one symbol. Symbol overload means that multiple constructs have the same symbol. Symbol excess means that the notation has symbols for which no construct is defined. However, a certain amount of *symbol deficit,* where language constructs have no symbol, can be used to reduce graphical complexity.

Perceptual discriminability deals with the distinguishability of the symbols. It encompasses the following subcategories: *Visual distance* provides a measure that can be computed from the number of visual variables in which symbols differ, and the number of perceptible steps of each different variable. The exact equation is not defined by Moody, he instead refers to relevant works "from psychophysics". It is thus left as an exercise to the person evaluating a visual notation. The *primacy of shape* demands that one visual variable, shape, is used before other visual variables to discriminate symbols. *Redundant coding* stipulates that more than one visual variable can be used to differentiate between symbols. *Perceptual popout* requires symbols to have unique values for at least one visual variable. *Textual differentiation* should be avoided.

Semantic transparency describes the ease of understandability of symbols. Symbols that are *semantically immediate* are intuitively understood by novice users. Symbols that are *semantically opaque* aren't, because the relationship between symbol and meaning is arbitrary. In between these two levels of transparency is *semantic translucency*, where symbols are not immediately understood, but provide good hints for memorizing them. In the worst case, symbols are *semantically perverse*, which means that the meaning suggested by the symbol is completely different from what it actually means.

Complexity management measures the amount to which mechanisms to deal with diagrammatic complexity are built into the notation. *Modularization* helps reducing diagrammatic complexity by constructs that facilitate chunking big diagrams into smaller ones. An example is the UML concept Package, that allows to hide the intrinsics of out of scope packages on diagrams while still showing the context of a particular package in scope. The language constructs facilitate modularization on a semantic level. This has to be accompanied by decomposition rules for diagrams, which support modularization on a syntactic level, e.g. rules when a package should be shown as a black (package symbol alone), grey (package symbol containing only some of its parts), or white box (package symbols containing all of its parts). *Hierarchy (levels of abstraction)* asks for language constructs that facilitate constructing hierarchies of diagrams. This partly overlaps with modularization if (de-)composition is used as the hierarchy mechanism. Another mechanism mentioned by Moody is summarization (abstraction), which we interpret as generalization where a more general class on a higher-level diagram can be exploded to a diagram showing the class hierarchy also including the specialized subclasses. This is something different than classification, where commonalities of objects are summarized into a class that can represent all these objects (i.e. the objects become

[7] We will refer to language constructs as concepts in this dissertation, or meta-classes when the focus is on the meta-model.

instances of the class).

Cognitive integration targets the problem of navigation between multiple diagrams of a model. The question of interest is whether a modeling language has intrinsic mechanisms to integrate diagrams. If a modeling language supports multiple types of diagrams, *homogeneous integration* (integration of diagrams of the same type) can be differentiated from *heterogeneous integration* (integration of diagrams of different types). Integration needs to be supported *conceptually*, where the language has mechanisms to aid the user in building a coherent mental representation of the whole system (e.g. root, summary, or context diagrams), and *perceptually*, where the language has mechanisms to keep the user oriented within the model (e.g. navigational links to other diagrams, or diagram labels).

Visual expressiveness is a measure on how many different visual variables are used within the notation of a language. The degree of visual expressiveness is determined by counting the visual variables that are used to differentiate between symbols. If none is used, a language is called *nonvisual*, if all eight known visual variables are used, it is called *visually saturated. The visual variables reported by Moody are horizontal position, vertical position, size, brightness, color, texture, shape, and orientation.*

Dual coding stresses that, while text should not be used as a primary means to differentiate between symbols, it is very effective in a supporting role. For example, the meaning of already well differentiable symbols is even easier to grasp with *hybrid symbols*, symbols that are also using textual information. *Annotations* can be used to facilitate the understandability of diagrams.

Graphic economy demands to limit *graphic complexity*, the number of different symbols in a notation - with six symbols as an upper limit. Moody argues that the existence of too many distinct symbols can reduce the effectiveness of the notation dramatically. Therefore, graphic economy has to be balanced with visual expressiveness. Many notations exceed that limit due to the semantic richness they strive for. An argument against the limit of six distinct symbols is a pragmatic one. Notations should be designed to support novice users in learning, but on the other hand, should not limit expert users (which the novice users will once become) in expressiveness too much. If every language had only six symbols, it could only focus on a limited extract of the problem at hand. The number of languages to express all aspects of a complex problem would grow. We therefore suggest further analysis whether this definition of graphic economy is really helpful.

Cognitive fit suggests to define visual dialects, with each dialect supporting different audiences, e.g. novice or expert users. For a requirements modeling language it might be helpful to create different dialects for different stakeholders, e.g. one for the customer, one for the business analyst, and one for the system developer. As already indicated in an earlier paragraph, the different principles interact with each other. For more details, we refer to (Moody 2009).

Moody's nine principles have been used to analyze the notation of UML (Moody and Hillegersberg 2009), i* (Moody, Heymans, and Matulevičius 2009), UCM (Genon, Amyot, and Heymans 2011), KAOS (Matulevičius and Heymans 2007), BPMN (Genon, Heymans, and Amyot 2011), and three instructional design languages (Figl et al. 2010). URML differentiates itself from these languages by explicitly following some of Moody's principles. An account of how the principles were applied in the design of URML's notation is given in section 4.3.3. Mäder and Cleland-Huang claim that Moody's principles were followed in the design of VTML (Mäder and Cleland-Huang 2010). Some authors are aware of the principles, but have not incorporated them yet, e.g. Grundy et al. (Grundy et al. 2013).

Caire et al. propose to involve "naïve users" in the notation design process (Caire et al. 2013). The term *naïve users*, also referred to by Moody (Moody 2009), however remains undefined. We assume that the meaning of the term, intended by Caire et al., is that *naïve users* are persons to which diagrams are presented, the diagrams being expressed in the notation of a graphical requirements modeling language[8], and the users having no prior knowledge about the notation and the underlying concepts of the modeling language. The hypothesis is that by involving *naïve users* in the design of symbols for the concepts of a modeling language, symbols with higher semantic transparency can be created. To test this hypothesis, Caire et al. undertake multiple steps.

In a first step, an experiment based on the sign production technique of Howell and Fuchs (Howell and Fuchs 1968) is performed. The basic assumption of that technique is "that subjects, when properly instructed, can actually produce signs for referent concepts and will do so in frequencies proportional to stereotype strength". Stereotype strength would be measured in another experiment, where participants would have to reproduce the meaning of a presented symbol. The assumption here is that "to the extent that a given stimulus evokes consistently the same or similar responses in a large segment of the communicating population, such a stimulus can be said to have meaning and, presumably, a considerable potential for transmitting information." To use Moody's terminology, the class of symbols that has been produced most often is considered to have a better semantic transparency than a class of symbols that has been produced less often. In Caire et al.'s experiment, 104 "undergraduate students in Economics and Management from the University of Namur" had to produce symbols for nine concepts of the i* modeling language.

In a second step, the symbols produced were classified by three judges, thus partitioned into categories of symbols. With the result of the classification, the category produced most often could be determined for each concept. From that category the judges chose the most representative symbol as the *stereotype* for the concept. For the other categories, the most representative symbol was also chosen in preparation of the third step.

In the third step, the assumption was that the previously identified stereotypes do not necessarily represent the symbol with the optimal semantic transparency. Another experiment was performed, in which 30 different students from the same university had to choose one of the category representants as the symbol that subjectively represents the concept best. The symbol with the most participant votes was chosen as the concept *prototype*. Only for three out of nine concepts, the participants of that experiment chose the stereotype of the previous experiment as the prototype. This seems to be in accor-

[8] In the context of Moody's paper, this would be generalized to any graphical modeling language.

dance with Jones (Jones 1983), who reported on experimental results from a comparable experimental setup, where the correlation between judges' rankings and stereotype could be confirmed, but not for all concepts. For some concepts, judges ranked symbols best that had only been proposed by one person (i.e. were no stereotype). To the best of our knowledge, there is no comprehensive theory to explain this phenomenon yet. The mismatch could be explained by the varying representativeness of the participants of the two experiments, with varying creativity of the participants, as suspected by Jones. It could also be connected to the varying ease of visualization of the different concepts. Ng et al. report that "the more visual the referent, the less difficulty the users had in illustrating it". (Ng, Siu, and Chan 2012)

Caire et al performed another experiment as a fourth step, a comprehension test to compare four different symbol sets for the given nine i* concepts. The symbol sets were the stereotype symbols, the prototype symbols, the original symbols created by the designers of i*, and symbols created by a design committee that had the goal of following Moody's principles. The participants of the test were 65 participants, "undergraduate students in Interpretation and Translation from the Haute Ecole Marie HAPS-Bruxelles or Accountancy from the Haute Ecole Robert Schuman-Libramont", randomly distributed to four experimental groups of approximately equal size. In each experimental group, the participant received a booklet containing nine pages. At the top of each page, a symbol from the symbol set was presented, and at the bottom, all concept names and definitions were presented in a table. The participants had to assign that concept to the symbol, which in their judgement was best represented by the symbol. It was recorded whether the participant could infer the correct concept from the given symbol. Based on the recorded data, a "hit rate" for a given symbol could be computed. Surprisingly, the stereotype symbol set had a higher mean hit rate than the prototype symbol set. The prototype symbol set only performed slightly better than the design committee symbol set. As expected by the experimenters, the original symbol set had the worst performance.

The problem we see with the sign production technique is that a single sign production experiment is not very likely to convey the most semantically immediate symbol for a given term. A major challenge is not only the representativeness of the experiment participants. Even for a series of such experiments, the experimenter will be unable to say that the ultimately best symbol has been found. Whenever the experimenters decide that no new symbols will occur, a certain probability remains that in a hypothetic next run, some participants would actually produce a better symbol (for whichever reasons, e.g. higher creativity, different cultural background). In theory, the experiment has to be repeated infinitely with changing participants, and only during a period where no fundamentally new symbols are produced, one can evaluate the stable symbol set. This leads to the argument that the experimenters need a set of criteria to decide when there are many enough, good enough symbols, which leads us back to, for instance, Moody's criteria. In our eyes, sign production experiments can be used to get started with notation design, but should not be the ultima ratio for all aspects of notation design, as not all influencing factors are already well understood. Besides that, it focuses on semantic transparency of the particular symbols, and does not take other criteria, as e.g. perceptual discriminability, into account.

2.5. Use Cases of a Graphical Requirements Modeling Tool

In contrast to natural language a requirements modeling language is not only used for face-to-face communication. To deal with the complexity of the requirements engineering process, several components are needed, which are outlined in Figure 2.8. The model repository is structured in terms of the abstract syntax meta-model and contains instances of the elements defined therein. The repository is accessed by modeling language users via a requirements engineering tool. The tool enables users to create expressions in the language, and work with the requirements knowledge that is already contained in the model repository. The tool can also offer support for requirements elicitation strategies, that provide guidelines for populating and elaborating a model. In this section, the main use cases of such a computer-aided requirements engineering (CARE) tool are described.

Any tool supporting a graphical modeling language has high-level use cases for the management of model elements and diagrams, and for performing queries on the model. These use cases are discussed in Section 2.5.1. The concept of model-based requirements elicitation strategy is explained in Section 2.5.2.

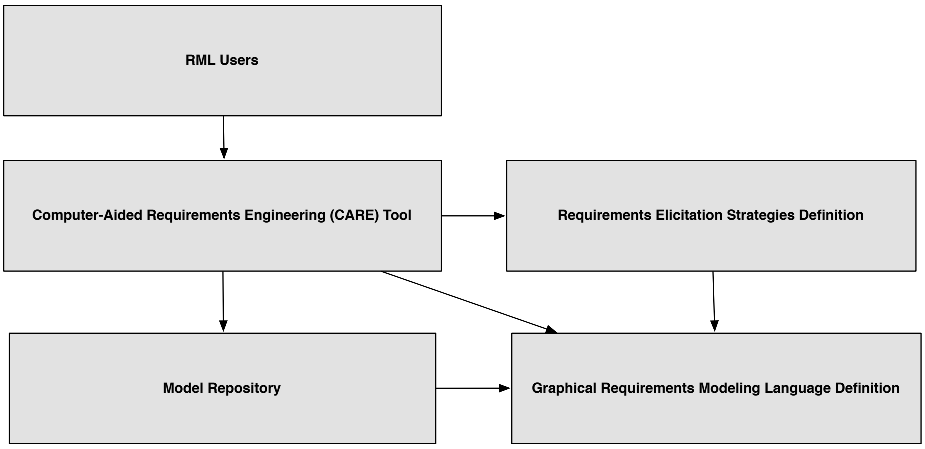

RML Users
Computer-Aided Requirements Engineering (CARE) Tool
Requirements Elicitation Strategies Definition
Model Repository
Graphical Requirements Modeling Language Definition

Figure 2.8.: Coarse architecture of a modeling language system

2.5.1. Basic Model Management

A tool implementation to support a modeling language must support the creation, modification and management of models. This entails working with model elements. A user creates elements, either instances of entity meta-classes, or instances of relationship meta-classes. When creating a relationship, two entities have to be selected and modified in order to add the relationship to their properties. Deleting an element removes it from the model, if the element is an entity, also the relationships in which it participated have to be deleted. If the element is a relationship, any elements participating in that relationship have to be modified. Beyond relationships, modification of an element does entail giving values to the attributes of the element. As some properties of model elements are not shown on diagrams, a representation of the element that includes these properties is needed. A URML model of these high-level use cases is shown in Figure 2.9. The URML syntax will be explained in Chapter 4. For this section, it is sufficient to assume that the diagrams can be interpreted like UML use case diagrams, only with a different symbol.

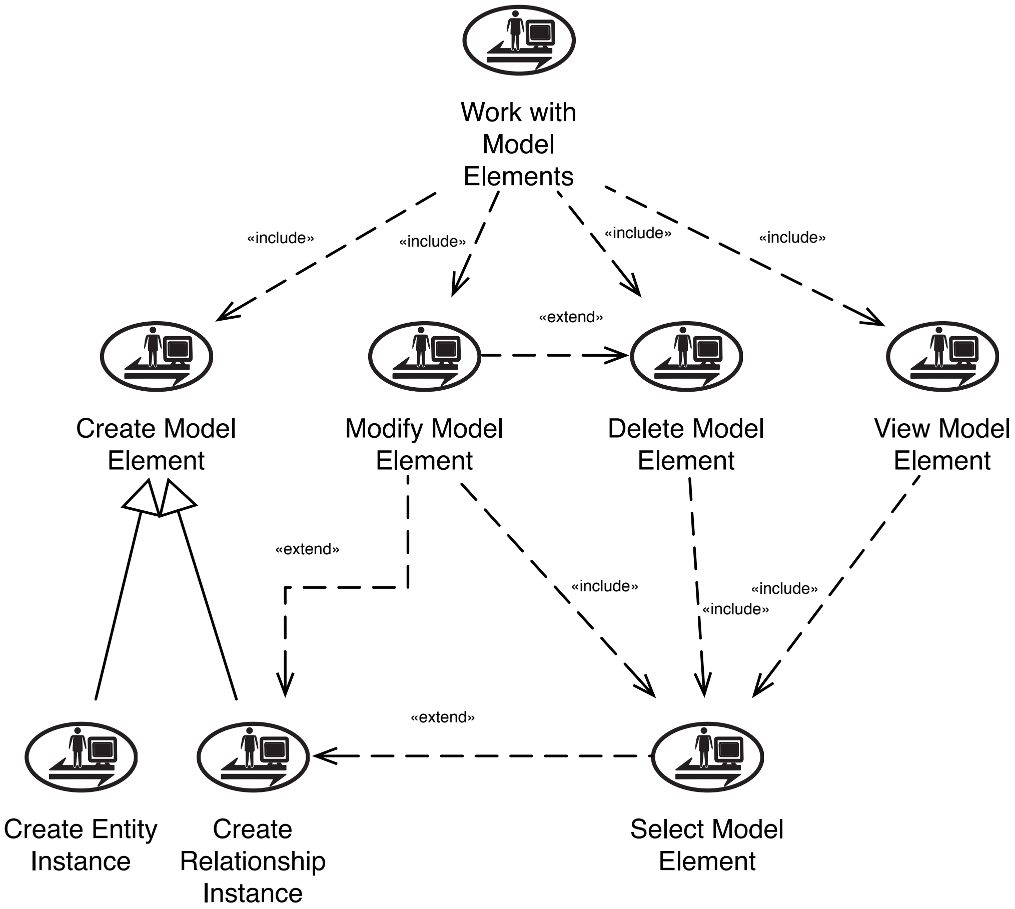


Figure 2.9.: Work with Model Elements high-level use case and included use cases.

Beyond working with elements, there are use cases to selectively browse the model (Fig. 2.10). A common mechanism for this purpose are diagrams. A diagram is a view on parts of the model. It contains elements and relationships that are spatially arranged on the diagram. After the creation of a diagram by the user, the diagram is empty. The user then adds entities and relationships. When an entity is added that is participating in relationships with entities already on the diagram, these relationships can be added automatically. If the user wants to focus only on some of these relationships, he may hide the unwanted relationships from the diagram. The layout of the elements on the diagram can be supported by graph layout algorithms or is done manually by the user. As the number of diagrams can grow large, there is a need of a navigation mechanism between the diagrams. If a diagram is not used anymore, it can be deleted.

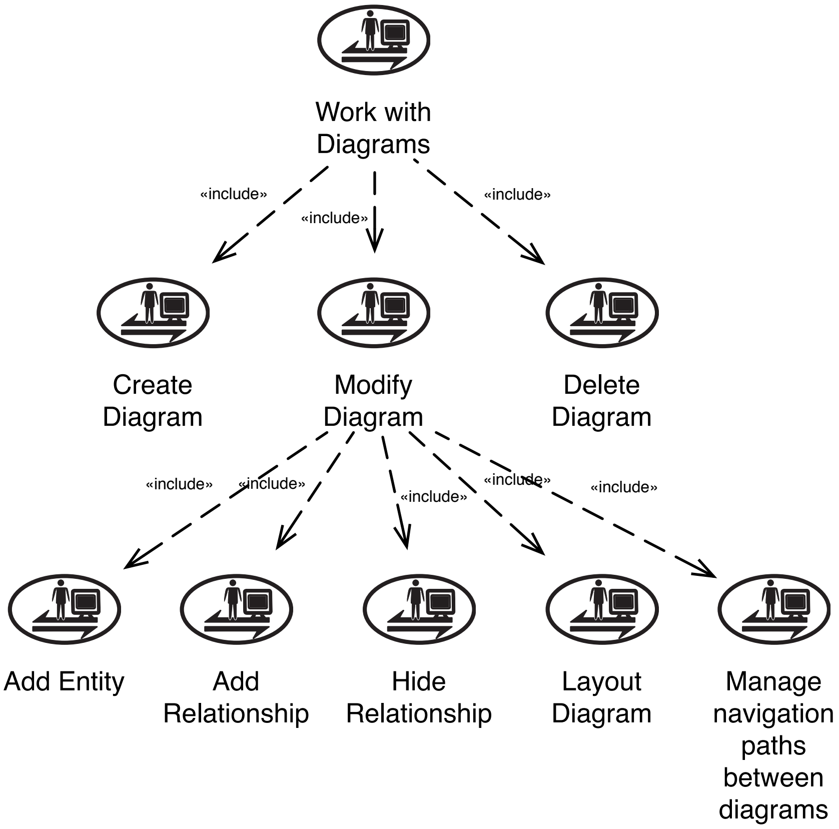


Figure 2.10.: Work with Diagrams high-level use case and included use cases.

To facilitate working with complex models, an RML tool provides mechanisms to work with model queries (Fig. 2.11). Users can formulate queries, which relate to meta-classes. Queries can be refined by defining constraints to only select meta-class instances with a given attribute set to a specific value. Complex queries can be constructed with logical expressions. When the user performs a query, the tool traverses the model and returns the model elements matching the criteria defined in the query. For users to digest the query results, the tool can present them in a list, on a diagram, or in a matrix. No matter which representation of search results was chosen, the user has the possibility to save a search for later re-execution. For any element in the search result the user can request a list of diagrams that the given element is on. The model query use cases support complexity management, as search results provide a quick entry to the model. They help avoiding redundancy in the model, as users can use query results to find out whether certain knowledge has already been modeled.

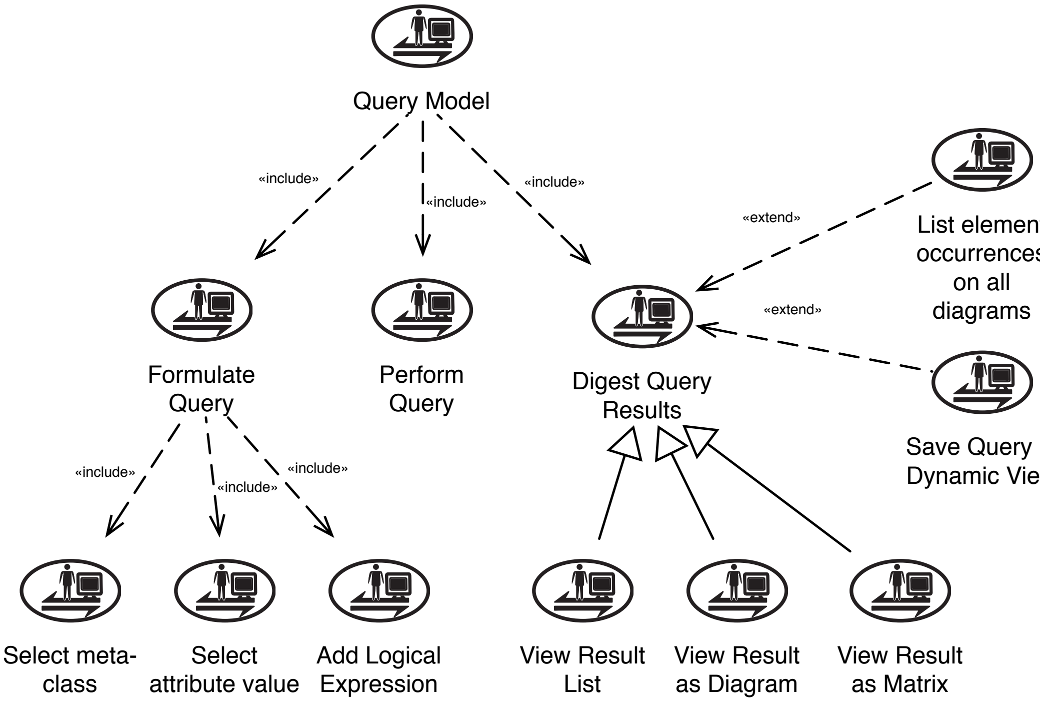


Figure 2.11.: Query Model high-level use case and included use cases.

All use cases discussed in this section are not language specific and realizations of them can be found in different CASE tools for different languages. We therefore do not discuss them in greater detail. Use cases that were realized in a reference implementation of URML are described in Chapter 6.

2.5.2. Model-Based Requirements Elicitation Strategies

Potts defines the term elicitation strategy as “a set of guidelines for identifying the correct sources of requirements and background information, eliciting requirements from them, and resolving conflicts among them” (Potts 1991). He states that elicitation strategy is the “most communication-rich” aspect of requirements analysis. Potts refers to CORE (Mullery 1985) as an exemplary elicitation strategy. In the description of CORE, Mullery delineates strategy from tactics: “strategy defines the general intentions of a plan”, whereas “tactics control the perturbing effects on the strategy of local disturbing influences".

This dissertation is not concerned with particular elicitation techniques as interviews or questionnaires and tactics when to apply which technique. In this dissertation, we are interested in the influence of the abstract syntax of a requirements modeling language on the requirements elicitation process. In particular, we take the stance that the abstractions of the ASM determine the knowledge that can easily and uniformly be recorded during elicitation sessions. If a stakeholder talks about certain matters she deems requirements-related, but the requirements modeling language offers no abstractions that are semantically close, the knowledge exposed to the requirements analyst cannot be recorded in expressions of the modeling language.

We are therefore interested in a special class of elicitation strategies introduced by Dardenne et al., called acquisition strategy in their terminology: "An acquisition strategy in this framework defines a well-justified composition of steps for acquiring components of the requirements model as instances of meta-model components" (Dardenne, Lamsweerde, and Fickas 1993). We call this *model-based requirements elicitation strategy.* Such a requirements elicitation strategy enforces a specific way to traverse a meta-model (Dardenne, Fickas, and Lamsweerde 1991). Tactics in that context provide patterns for model elaboration (Darimont and Lamsweerde 1996). Model-based requirements elicitation strategies work with any language meta-model, but the contents of the meta-model determine to a great extent which strategies and tactics are available. For example, a use-case driven strategy could not be used with a language not containing a UseCase abstraction, or has to be adapted if that is possible. Therefore, the richer the ASM of a requirements modeling language is, the more flexibility it offers regarding the application of elicitation strategies.

Goal URML supports multiple model-based requirements elicitation strategies.

Existing model-based requirements elicitation strategies are discussed in Section 3.5.

3. State of the Art

In this chapter, graphical modeling languages that include requirements or include concepts that are close to requirements are presented. The two most significant languages in this regard are KAOS and URN. KAOS is described in Section 3.1 and URN in Section 3.2. VLML is a visionary language to focus on the early requirements phase. It is described in Section 3.3. In Section 3.4, we refer to modeling languages that are not entirely dedicated to early requirements, and to approaches that do not propose a language on their own, but suggest a combination of certain requirements-centric concepts in models.

3.1. KAOS

KAOS is a goal-oriented graphical requirements specification language (Dardenne, Lamsweerde, and Fickas 1993). The acronym originally stood for "Knowledge Acquisition in automated specification" and was later changed to "Keep All Objectives Satisfied" (Lamsweerde 2003). A comprehensive description of the language is given in (Lamsweerde 2009). As the meta-model has changed considerably over the years, the description of the language in the following paragraphs is based on that textbook[1] and not on the original articles in which the language was first presented.

The KAOS meta-model consists of five integrated parts. The *goal meta-model* provides concepts for modeling goals and their refinement, goal conflicts, and obstacles towards goals. The *agent meta-model* provides abstractions for modeling environment and software agents, and what is expected or required of them. The *operation meta-model* focuses on the operationalization of requirements, events that trigger operations, and the data flow of operations. The *object meta-model* enables modeling the structure of conceptual objects in order to capture domain properties and to define concepts, that can be referred to by the other parts of a KAOS model. The *behavior meta-model* provides abstractions to model the required behavior of agents. The KAOS meta-model has no root abstraction to describe the commonalities of all meta-classes, though it is stated that all meta-classes share the mandatory attributes `name` and `def`, and the optional attribute `issue`. We only discuss their meaning for a given meta-class wherever they were made explicit in the meta-model.

In the next subsections, the five parts of KAOS are described and discussed: Subsection 3.1.1 concerns the goal meta-model, Subsection 3.1.2 the object meta-model, Subsection 3.1.3 the agent meta-model, Subsection 3.1.4 the operation meta-model, and Subsection

[1] As a convention for this section, text put in quotes is a literal citation from (Lamsweerde 2009). We follow this convention only for larger phrases and not for keywords, e.g. goal meta-model.

3.1.5 the behavior meta-model. Each subsection explains the meta-classes contained in the respective meta-model part, and the according notational elements are shown by example. Where we have identified issues with the meta-model, we will discuss them in the appropriate subsection. To explain some of these issues, we have to refer to specific sections of van Lamsweerde's book in order to highlight consistency issues between written text and meta-model.
A discussion of general issues is provided in Subsection 3.1.7.

3.1.1. Goal meta-model

The central abstraction of the goal meta-model (Fig. 3.1) is the `Goal`, which is defined as "a prescriptive statement of intent that the system should satisfy through the cooperation of its agents" (Lamsweerde 2009, p.266). As mandatory attributes, every `Goal` has a unique `name` and a `def` attribute to hold an informal textual definition of the `Goal`. The other attributes are optional. The `category` attribute allows to classify `Goals` following a taxonomy that is external to the meta-model, and orthogonal to the `Goal` taxonomy present in the meta-model. The external taxonomy should be defined as an enumeration in the meta-model, we assume it has been omitted from the meta-model, as it is defined in a dedicated figure earlier in the book. The main purpose of that taxonomy is to differentiate between functional and non-functional `Goals`. The `priority` attribute can support conflict resolution and the choice of alternative `Goals`. The `stability` attribute can be used for "change anticipation" and model reviews. It indicates how likely it is that the knowledge about a `Goal` changes. The `source` attribute allows to informally specify where a `Goal` comes from, e.g. which stakeholder mentioned it, or during which session it was discovered. The `issue` attribute supports model reviews. It allows to indicate that there is an issue with the given `Goal`, and provides a textual description to detail the issue.

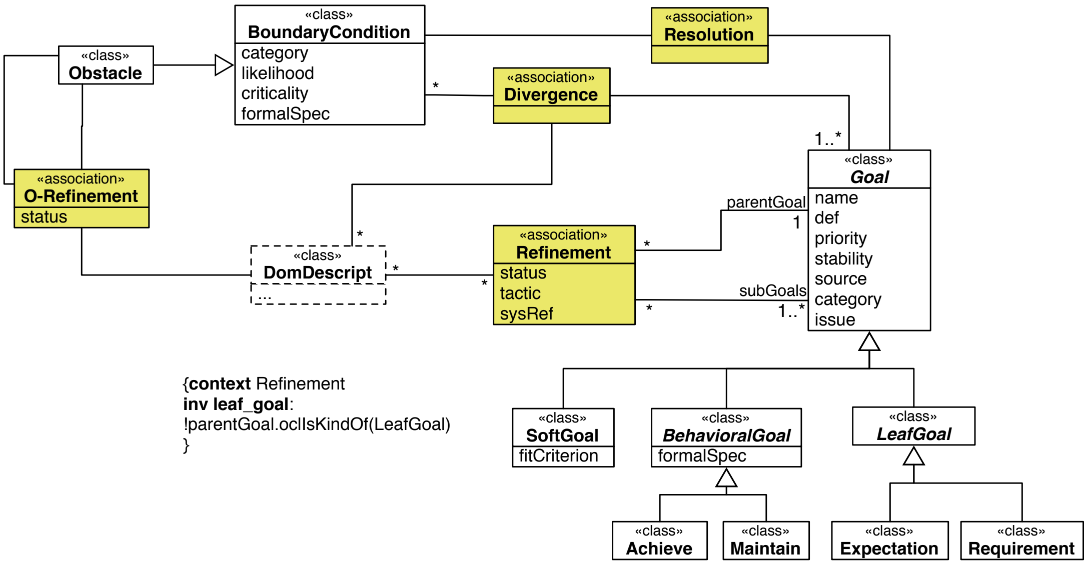


Figure 3.1.: KAOS Goal Meta-Model. Adapted from (Lamsweerde 2009, p.488), with modifications.

It is not clearly stated whether Goal is abstract, so it could be assumed that instances of Goal can appear on diagrams. Van Lamsweerde however clearly states that “a goal is either a behavioral goal or a soft goal” (Section 7.3.1). We therefore consider the Goal meta-class being abstract, even though the previous statement is contradicted by the presence of a third Goal subclass in the meta-model, the LeafGoal. So the meta-model would actually suggest that a goal is either soft, behavioral, or leaf. To facilitate an analysis of this situation, we first discuss the subclasses of Goal, and the Refinement meta-relationship, and then make a proposal how the meta-model could be improved.
BehavioralGoals prescribe intended system behavior: behavioral Achieve goals prescribe behavior that aims at finally fulfilling a target condition and behavioral Maintain goals prescribe behavior that aims at maintaining a condition. LeafGoals represent Goals that are not further decomposable and can be satisfied by a single agent. The relationship of agents to goals is part of the agent meta-model and is described in Subsection 3.1.3.
There are two kinds of LeafGoals: A Requirement is a fine-grained goal that can be satisfied by a single agent of the software to-be. An Expectation is a fine-grained goal for which an agent of the system's environment is expected to satisfy it. As neither Achieve, nor Maintain, Expectation, or Requirement have different attributes or associations as compared to their superclasses, it would be preferable to model these aspects of the respective Goal subclass via attributes typed by an enumeration.
SoftGoals establish criteria for choosing between alternative system behaviors. A criterion introduced by a SoftGoal is represented via the fitCriterion attribute. SoftGoals play a special role when discussing the decomposition of BehavioralGoals, when

a given BehavioralGoal has multiple alternative refinements. In such a case, every possible refinement can be evaluated with respect to its contribution to the "satisficing" (Mylopoulos, Chung, and Nixon 1992) of a SoftGoal. This aspect is not explicit in the meta-model, but has been described in the according paragraphs of van Lamsweerde's book (especially in Figure 8.16 of Section 8.8.2). The assignability of SoftGoals to agents (assignment and responsibility of Agents is discussed in Section 3.1.3 of this dissertation) is unclear. It is also unclear whether the decomposition of SoftGoals is realized through instantiation of the same Refinement relationship as the one used for BehavioralGoals. SoftGoals are decomposable, but it is unclear whether SoftGoals may only be refined by SoftGoals, or also by BehavioralGoals. The next paragraphs therefore discuss the Refinement relationship.

Every goal is part of a hierarchy, which is a directed graph of goals and refinement links. A Refinement identifies a set of subgoals that contribute to a common parent goal. A sub-goal can be part of different refinements (and thus possibly contributes to more than one parent goal) and a parent goal can have multiple alternative refinements. The decomposition of one goal into a set of subgoals is also called *AND-refinement*, because the parent goal can only be considered satisfied when every subgoal is satisfied. If a goal has multiple alternative AND-refinements, this is called *OR-refinement*, because only one of the alternative AND-refinements can be selected for the system under discussion. Our representation of the goal meta-model in Figure 3.1 is a simplification of the original meta-model regarding Refinement, as it is represented by van Lamsweerde (Lamsweerde 2009, p.488). An excerpt of the original model is shown in Figure 3.2. In this model, Goal is either related to Refinement via an OR-Ref or an AND-Ref association. In addition, there is an unnamed association between Refinement and DomDescript. Between this unnamed association and AND-Ref, there is a dotted line, which, according to UML, means that there is a constraint concerning the two associations. According to van Lamsweerde, this is called an OR-association. The intended meaning in Figure 3.2 is that "domain properties may be involved in the refinement as well"[2].

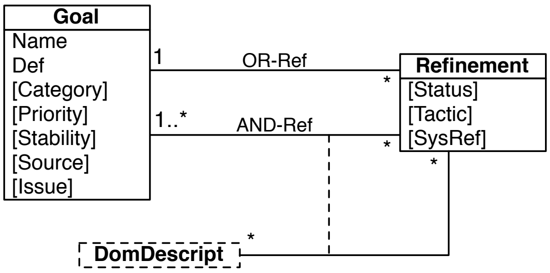


Figure 3.2.: KAOS Goal Meta-Model excerpt from (p.488).

Figure 3.3 shows a possible instantiation of that Refinement-related part of the original goal meta-model. It shows that Goal1 has two alternative AND-refinements. The first refinement entails Goal2 and DomainProperty2, the second entails Goal3 and Goal4. The figure could be interpreted as that there is one instance of OR-Ref, represented by the upper two arrows, two instances of Refinement, represented by the two circles with yellow background. Given these interpretations, there is, however, a difficulty in un-

[2] The DomDescript meta-class belongs to the object meta-model and thus will be explained in detail in Subsection 3.1.2.

derstanding how AND-Ref and the unnamed association are instantiated. As AND-Ref also relates Goal to Refinement, it seems there are two instances of AND-Ref, the first relating Goal2 to Refinement 1, and the second relating Goal3 and Goal4 to Refinement 2. The first AND-Ref is then combined with an instance of the unnamed association to relate DomainProperty1 to Refinement 1.

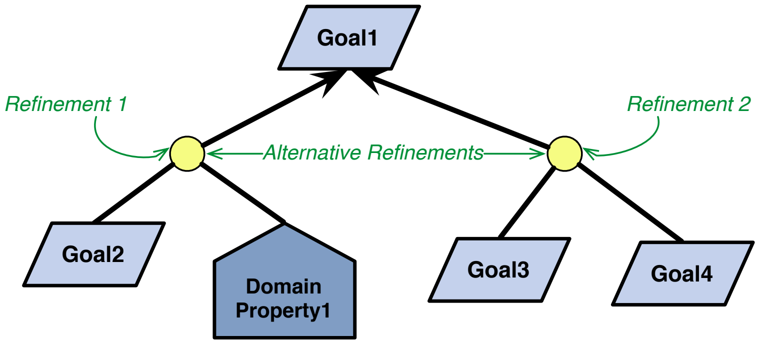


Figure 3.3.: Example instantiations of Goal, Refinement, and DomDescript.

In our re-interpretation of the meta-model part concerning Refinement (Fig. 3.4), we have focused on interpreting Refinement as a relationship meta-class. It also expresses, that a Goal can have multiple alternative Refinements, that a Refinement entails at least one Goal but can also entail many, and that optionally, DomDescript instances can be added to a Refinement. With the changed meta-model, the OR-association from above is not necessary anymore. It makes the above figure easier to interpret: We see two alternative Refinements of a Goal, and each Refinement is understood as a whole. It does not make much sense to consider the graphical constituents of the refinement notation (e.g. the yellow circles) in isolation. Through the naming of the association ends ending at Goal, it is also clearer that Refinement is used to build a goal tree. The refined meta-model also makes it easier to express a constraint on LeafGoals, which is not expressed explicitly in the meta-model diagrams: A LeafGoal must not be a parentGoal in Refinement relationships, i.e. LeafGoal instances can only be found at the bottom of a goal tree.

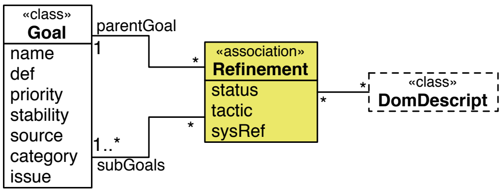


Figure 3.4.: Re-interpretation of Refinement-related meta-model.

In Figure 3.5, we present what could be an improved meta-model for KAOS, which refinement trees of behavioral goals more precisely.

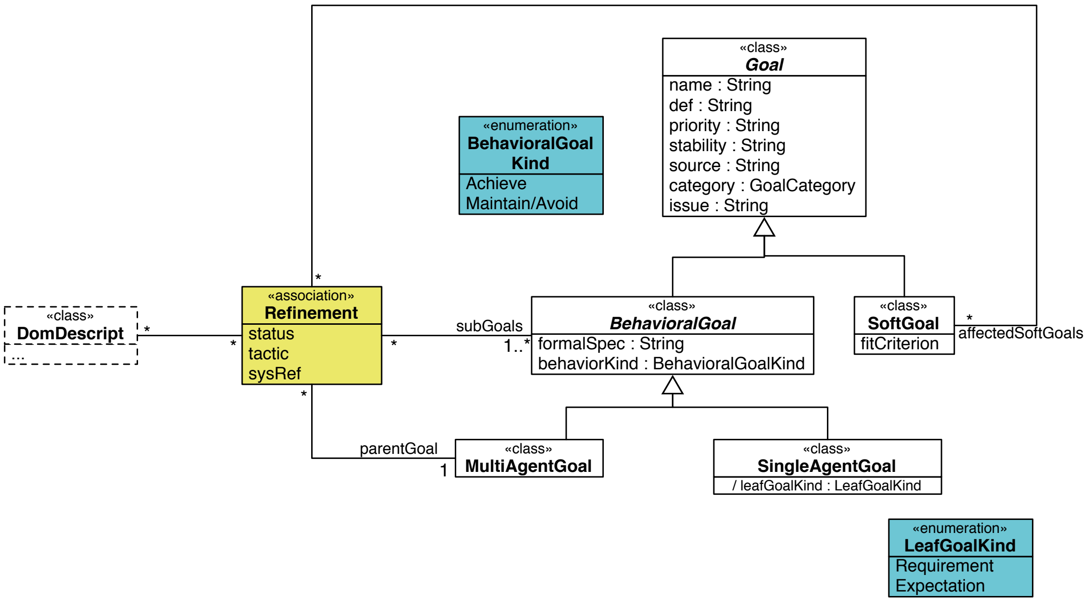


Figure 3.5.: Alternative model for the KAOS goal taxonomy and refinement trees.

Obstacle is a subclass of BoundaryCondition, and is described as “a precondition to the non-satisfaction” of an assertion, i.e. a Goal or a DomDescript. Obstacles are important for safety and security analysis, showing under which conditions a system can not achieve its goals. The category attribute of BoundaryCondition allows for classification of BoundaryConditions, and thus also Obstacles, as hazard, threat, dissatisfaction, misinformation, inaccuracy, and unusability. Hierarchies of obstacles can be modeled with O-Refinements, which are like the Refinement relationship, but dedicated to Obstacles.

Different goals have the potential to be in *conflict*, under certain conditions. Potential conflicts are expressed by the Divergence relationship. It relates at least two possibly conflicting Goals, and the BoundaryCondition under which the conflict manifests itself. The Divergence relationship has an alternative meaning: A special case of divergence occurs when Obstacles *obstruct* goals or domain descriptions. The Divergence relationship is hard to understand from the original meta-model (see Fig. 3.6), as depending on its actual intended meaning, different constraints are to be enforced in the model.

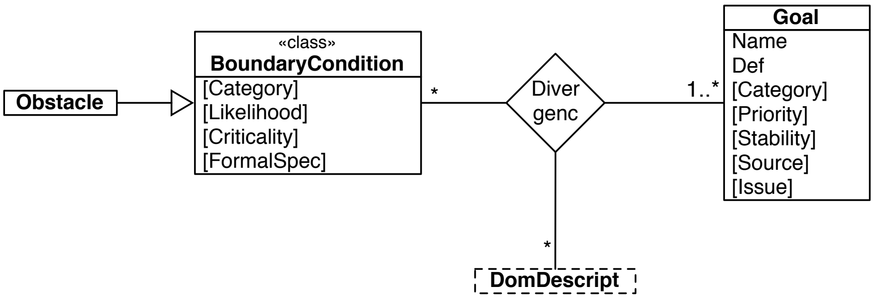


Figure 3.6.: Original `Divergence`-related meta-model, extracted from (Lamsweerde 2009, p.488)

In the case of a conflict, `Divergence` relates two or more `Goals`, i.e. the cardinality on the `Goal` must be 2..* instead of 1..*. In the case of *goal obstruction*, `Divergence` relates exactly one `Obstacle` with exactly one Goal. The exact interpretation of the * cardinality at the `DomDescript` end of the `Divergence` relationship is unclear. It is stated that `DomDescript` can only participate in the case of *domain description obstruction*, but not whether exactly one `Obstacle` obstructs exactly one domain property. Also not clear is how the * cardinality at the `BoundaryCondition` end is to be interpreted. In the case of goal obstruction it must be 1, but it is not clear if the `BoundaryCondition` is optional in the other two cases.

Modeling obstruction helps finding new `Goals` that are able to resolve the undesired condition. Resolving `Goals` are related to `Obstacles` via `Resolution` relationships. Van Lamsweerde only refers to `Resolution` when discussing obstacle diagrams, therefore it is not clear how a `Resolution` works in the case of goal conflicts. `Resolution` links have the same visual syntax as obstruction links (i.e. instances of `Divergence`), when connecting `Goal` and `Obstacle` instances, which is a case of symbol overload. In a tutorial on KAOS (Respect-IT 2007), the arrowheads of obstruction and resolution links have different color fills.

It is unclear where a resolution link would be attached to, if one `Goal` resolves the conflict of two other `Goals`, as `BoundaryCondition` has no graphical representation, a case of symbol deficit, and conflict links (also instances of `Divergence`) directly connect the conflicting `Goals`.

3.1.2. Object meta-model

The object meta-model (Fig. 3.7) enables defining concepts within and properties of the domain in which the future system will be situated. The object model view of a KAOS model provides the basic domain terminology that other views refer to.

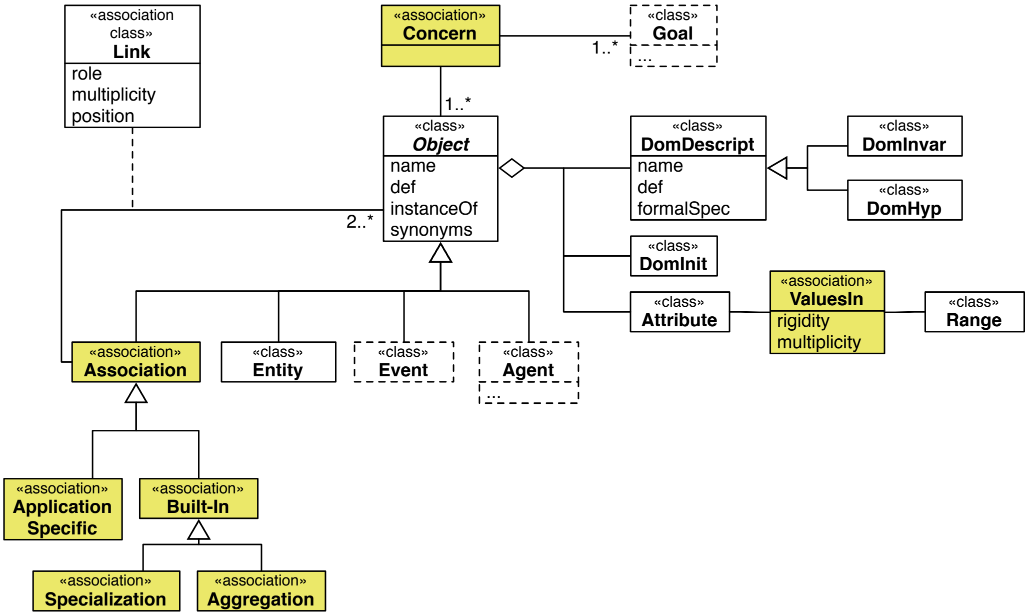


Figure 3.7.: KAOS Object Meta-Model. Adapted from (Lamsweerde 2009, p.489).

The central abstraction is the `Object`, representing a set of instances of domain objects that are shared between the system under discussion and its environment. This is an important difference to the semantics of an UML class, where the instances are objects that are part of the system under development and not necessarily shared with the environment. The `name` of an `Object` uniquely identifies it. The `def` property gives a textual definition of the `Object` in natural language.
The `instanceOf` property is hard to understand. The text accompanying the object meta-model states that it is a boolean attribute. The chapter discussing object models, however, does not mention it among the properties of the Object meta-class. In contrast, section 10.1.2 discusses a semantic relation *InstanceOf (o,Ob)* that evaluates to true of o is an instance of object ob. It is not clear how exactly the two concepts are related and under which circumstances the `instanceOf` property has the value `true`. `Objects` contain domain descriptions (`DomDescript`), `Attributes`, and initialization statements (`DomInit`). As multiplicities were omitted for that part of the object meta-model, we assume any `Object` can have zero to many of these.

An `Attribute` describes an internal feature of an `Object`, it has a `name` and a `Range` of values. The `name` property is not mentioned in the object meta-model, but in the according paragraphs of the book. Depending on the properties of the instantiated `Range`, the `Attribute` is called an elementary or structured `Attribute`. It is called *elementary* if the `Range` is a set of atomic values, e.g. `Integer`. It is called *structured* if the `Range` “is defined in terms of standard sort constructors such as Tuple, SetOf, SequenceOf or Union”. The object meta-model provides no mechanism to refer to these terms, therefore our representation of the object meta-model in Figure 3.7 is as vague as in the book. An `Attribute` is linked to a `Range` via the `ValuesIn` meta-relationship. In the book it is modeled as an association class, but we found it simpler to model it as a class. The `multiplicity` and `rigidity` properties of `ValuesIn` define characteristics of the `Attribute` regarding the `Range`. The `multiplicity` property indirectly defines whether the `Attribute` is elementary or structured, for example, if the `multiplicity` is [1..1] or [0..1], the `Attribute` has to be elementary. The `multiplicity` states whether the `Attribute` is *optional* or *mandatory*, for example, if the `multiplicity` is [*] or [0..1], the `Attribute` is optional. The `rigidity` property of `ValuesIn` indicates whether the value of an `Attribute` stays constant over time. Rigid `Attribute` declarations are prefixed by the # sign. The object meta-model specifies no multiplicities of the association between `Attribute` and `Range`. We assume that each `Attribute` has a `Range`, and that a `Range` can be reused for the definition of multiple `Attributes`.

Domain descriptions, represented by instances of the `DomDescript` class, are descriptive statements about the phenomena shared between environment and system under discussion. Domain descriptions have a short `name`, a natural language definition (`def`) and, optionally, a formal specification (`formalSpec`). In terms of the object meta-model, a `DomDescript` instance makes a statement via its `def` or `formalSpec` properties, referring to the `Attributes` of an `Object` instance. If it makes a statement about multiple `Objects`, it needs to be attached to an `Association` that links these `Objects`. If a domain description is "expected to hold invariably regardless of how the system behaves", it is classified as a *domain invariant* (`DomInvar`). Domain invariants are also called domain properties throughout the book. If a domain description is subject to change, it is classified as a *domain hypothesis* (`DomHyp`). It is unclear why the `DomDescript` class is not abstract, as instances of `DomDescript` should bei either of type `DomInvar` or `DomHyp`.

An initialization, i.e. an instance of the `DomInit` meta-class, specifies how the `Attributes` of an object instance are initialized in the moment of instantiation. It seems odd that it has no properties in the meta-model. We would have expected properties comparable to the `def` or `formalSpec` properties of `DomDescript`.

Six properties of `Object` that are not mentioned in the object meta-model may be used in textual model annotations: `type`, `synonyms`, `issue`, `domInvar`, and `init`. As KAOS prescribes that UML class diagram notation is used for the object model, all objects are represented by rectangles and thus look very similar. Therefore, van Lamsweerde also suggests to add the `Object's type` as a textual annotation to the model. We have also not added a `type` attribute to `Object` in Figure 3.7, as the type can also be derived from the meta-class name. A `synonyms` property provides a list of strings that can be used as synonyms to the `Object's name`. We found no reason why it was not included with the object meta-model and therefore included it in Figure 3.7.

The `issue` property would, in our opinion, justify the introduction of a new class to the KAOS meta-model. According to van Lamsweerde, it is an optional attribute of any meta-class. Providing an `Issue` class, which might be subclassed to refer to specific other meta-classes or to a special abstract class that represents everything annotatable by an issue, could simplify queries regarding model issues.

The `has` property provides a list of textual definitions, one for each `Attribute`. If `At-`

tribute had a def property (as Object and DomDescript have), the has property of Object could be synthesized from these. As an Attribute has a name as well and that property is also not presented in the object meta-model, we assume the def property might also have been omitted.
The domInvar property is a derived attribute that picks the domain descriptions (i.e. from the contained DomDescript instances of the Object) that are an instance of DomInvar. In the model annotation the name and the def property of each DomInvar are presented. We have omitted the properties has, domInvar, and init from Figure 3.7, as they are derived, and to avoid redundancy on the diagram.

While Object is presented as a concrete class in the object meta-model, van Lamsweerde states that "any object instance is either an entity, association, agent or event instance". We have therefore interpreted Object as an abstract meta-class in Figure 3.7. Four concrete kinds of Object are supported: Entity, Event, Agent, and Association.
Entities are "autonomous and passive", i.e. they can exist independently of other objects, but do not control any behavior. Entity does not further specialize Object. Agents are "autonomous and active", i.e. they can exist on their own and entail behavior. Events are instantaneous objects that only exist "in a single state of the system". Agents and Events have more properties, but in this subsection, we are only interested in them being a subtype of Object. Further properties of the Agent meta-class are detailed in Subsection 3.1.3, and of the Event meta-class in Subsection 3.1.5.
Associations are non-autonomous, as they are dependent on the Objects that they link. An Association can link two or more Objects. It is called binary, if it links exactly two Objects, and n-ary if it links more than two Objects. The Link association class is intended to provide information regarding the role and the multiplicity of the participating Objects. Notation-wise, the role and multiplicity properties of Link are then placed on the the end of the association link. However, the Link association class, as shown in the meta-model, only provides enough meta-information for one association end, if its attributes are elementary. As the object-meta model dos not specify types for the properties, we can only guess how exactly they can be used. We assume that the intention of the meta-model is that the Link meta-class is instantiated as often as the association has ends; which end shall be annotated by the multiplicity and role properties could be defined by the position property, for which we found otherwise no explanation. The Link meta-class is particularly confusing because, throughout the book, instances of Association are also called "link". A meta-model similar to the UML meta-model seems more appropriate. In UML's meta-model, an additional meta-class Property is between Class and Association, thus facilitating that multiplicity and role can be specified per association end.

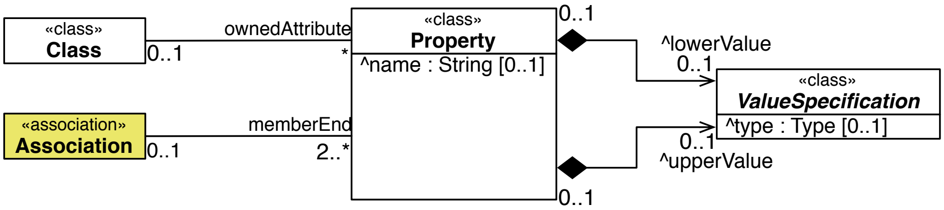


Figure 3.8.: UML Meta-Model excerpt, showing how multiplicities are modeled. Adapted from (OMG 2013b).

The object meta-model provides a small taxonomy of `Associations`, it supports `ApplicationSpecific` and `Built-In` associations. `ApplicationSpecific` is to be specified by a user of KAOS, it is, however, unclear to us how exactly the extension mechanism shall work. `Built-In` should be an abstract class, as `Built-In` instances are either instances of `Specialization` or `Aggregation`. As `Specialization` is a binary association, we assume the according constraint to limit the number of `Objects` it can link to two has been omitted from the object meta-model.

3.1.3. Agent meta-model

The central abstraction of the agent meta-model (Fig. 3.9) is the `Agent`, which is an "autonomous and active" subclass of `Object` - as defined in the previous section. `Agents` share the basic features `name`, `def`, and `load`, `name` and `def` being inherited from `Object`. The `load` attribute supports a high level load analysis. Unfortunately, it is not explained any further, especially its type is not mentioned. From the context in which it is mentioned, we assume that it is a derived attribute that reflects the number of responsibilities that an agent has. An additional attribute, `category`, is mentioned in Section 11.1 of van Lamsweerde's book (Lamsweerde 2009, p.396)[3]. The `category` attribute is described having an enumeration type, which includes the values `New Software Agent`, `Existing Software Agent`, `Device`, and `Human agent`.

[3] The `category` attribute is, however, not present in the section specifying the KAOS meta-model, therefore it is not shown in Fig. 3.9.

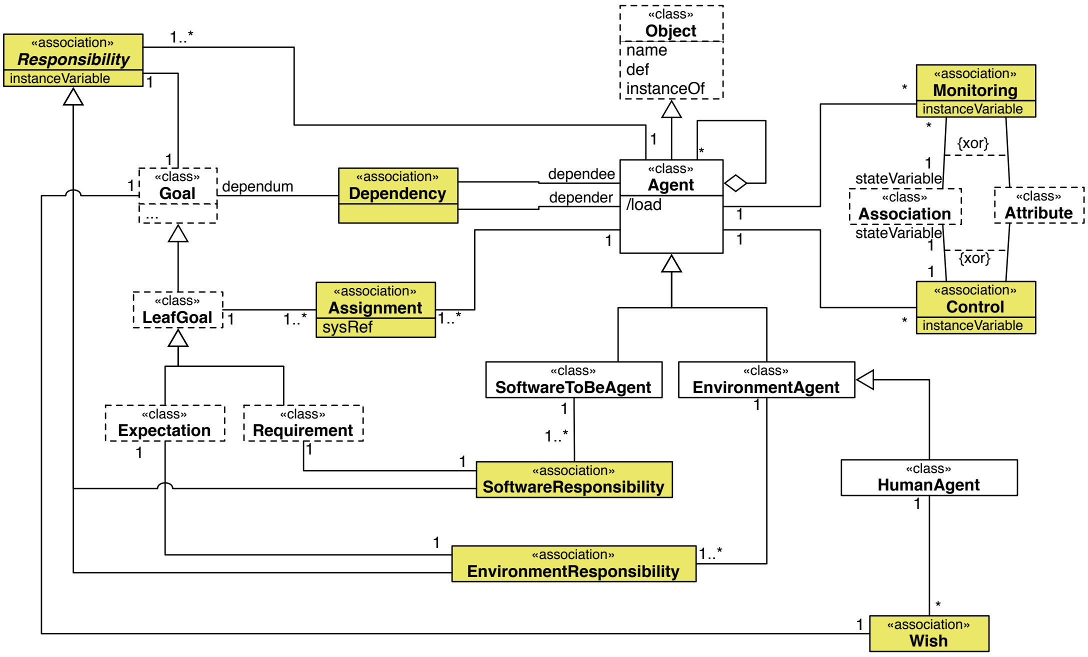


Figure 3.9.: KAOS Agent Meta-Model. Adapted from (Lamsweerde 2009, p.489-490).

An `Agent` is further specialized to two subclasses, `SoftwareToBeAgent` and `EnvironmentAgent`. The agent taxonomy is partially redundant to the `category` attribute: `SoftwareToBeAgents` can only have the `category` `New software agent`, whereas `EnvironmentAgents` can be either `Existing software agents`, `Devices`, or `Human agents`. A later section suggests that, instead of the category `Human agent`, there should be a subclass of `EnvironmentAgent` called `HumanAgent`. It becomes necessary with the introduction of the `Wish` meta-relationship, which relates exclusively `HumanAgents` to `Goals`[4]. The precise semantics of the `Wish` meta-relationship is not specified: We assume it means that an instance of `HumanAgent`, i.e. some real person that was participating in a requirements elicitation session, has expressed his or her wish that a certain `Goal` has to be satisfied. Alternatively, it could mean that all persons that are in that role (or a majority of them) have expressed that `Wish`.

The difference between `SoftwareToBeAgents` and `EnvironmentAgents` is the kind of `Responsibility` that they can assume: `SoftwareToBeAgents` are responsible for `Requirements`, whereas `EnvironmentAgents` are responsible for `Expectations`.

We have had some trouble interpreting the agent meta-model with respect to the details of the `Responsibility` meta-relationship. Due to van Lamsweerde's description of the agent meta-model, "the *Responsibility* meta-relationship is defined pairwise among specializations of Agent and LeafGoal". The corresponding excerpt from the meta-model is shown in Fig. 3.10.

[4] We did not include the two other categories of Agents in the diagram of Fig. 3.9, as the book states no special properties about them.

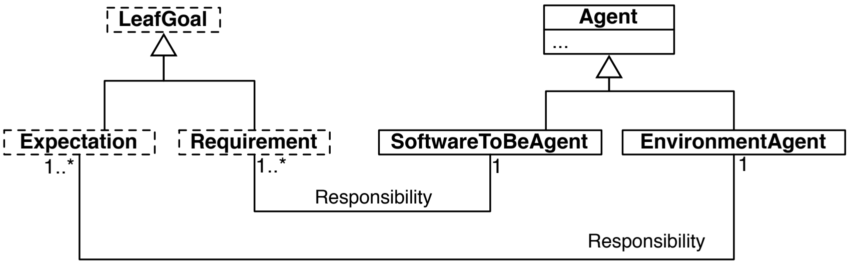


Figure 3.10.: KAOS Agent Meta-Model. Adapted from (Lamsweerde 2009, p.489-490).

In our eyes, it is misleading that the same class appears on a diagram twice. We saw three possibilities to interpret this. In the first interpretation, there are two separate relationships, one for the responsibility of `SoftwareToBeAgent` for `Requirements`, and one for the responsibility of `EnvironmentAgent` for `Expectations`. Due to the agent meta-model, both relationships are different as they connect different classes. However, in a later paragraph it is stated the `Responsibility` relationship has an attribute `instanceVariable`, which contradicts the interpretation that the two relationships are completely separate. To stick with the first interpretation, an abstract class is introduced to provide the common attribute (Fig. 3.11).

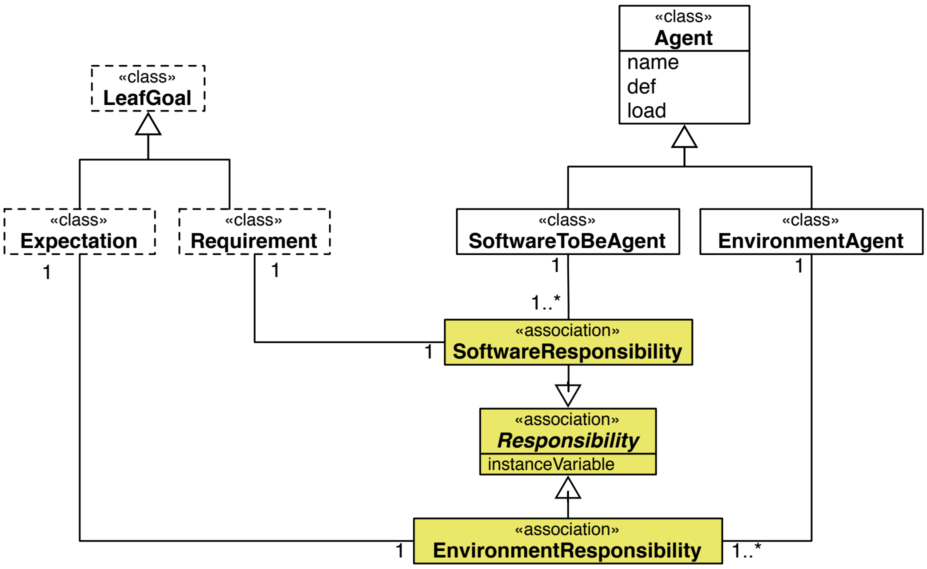


Figure 3.11.: First interpretation: There are two kinds of `Responsibility` relationships. Adapted from (Lamsweerde 2009, p.489-490).

In the second interpretation (Figure 3.12), there is only one `Responsibility` relationship with four associations. To avoid that it can be instantiated with four related elements, there is an attached constraint stating that if the related `Goal` subclass is an Expectation, the related Agent subclass has to be an EnvironmentAgent, and, if the related Goal subclass is a Requirement, the related Agent subclass has to be a SoftwareToBeAgent.

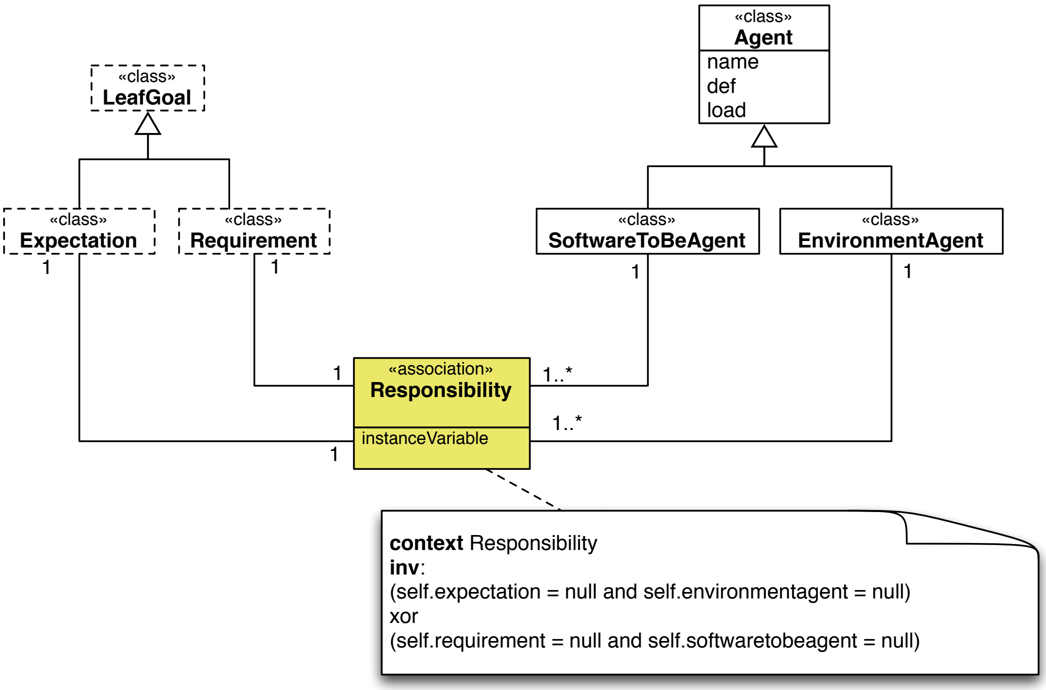


Figure 3.12.: Second interpretation: There is only one kind of Responsibility relationship, with a non-trivial constraint. Adapted from (Lamsweerde 2009, p.489-490).

Due to Section 11.4, however, Responsibility is in general a relationship between Agents and Goals. That statement is in conflict with the agent meta-model, that only presents Responsibility links between specific Agent and LeafGoal subclasses. Similarly, the previous two interpretations of that meta-model presented above are not consistent with the statement.

That inconsistency is fixed in Figure 3.13. The Responsibility meta-class, which was interpreted as abstract class in Fig. 3.9, becomes a concrete relationship between Agent and Goal, being subclassed by the more specific responsibility relationships between SoftwareToBeAgent and Requirement, and EnvironmentAgent and Expectation. The association ends between the subclasses of Goal, Responsibility, and Agent redefine the association ends between the Goal, Responsibility, and Agent classes, in order to avoid that SoftwareToBeAgents can be responsible for anything different than Requirements and that EnvironmentAgents can be responsible for anything different than Expectations. We stick to this third interpretation, as it is consistent with the statements made in (Lamsweerde 2009), under the assumption that the parent Responsibility meta-class has been omitted in Fig. 14.5 of the book.

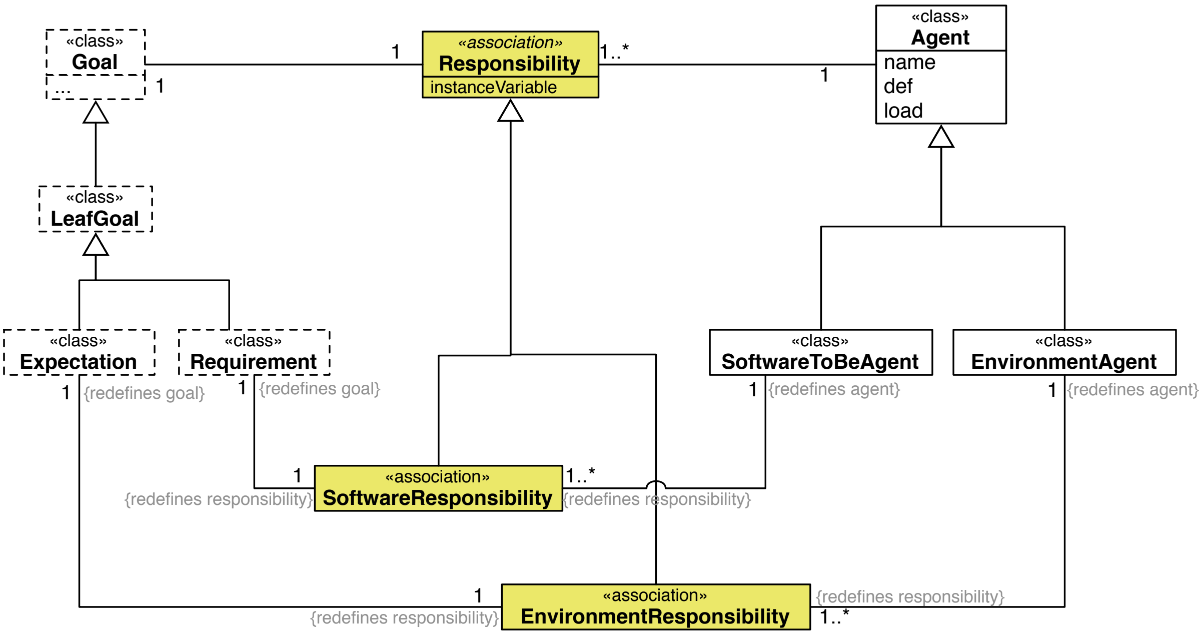


Figure 3.13.: Third and final interpretation: there are three kinds of `Responsibility` relationships. The association ends of the associations between the subclasses redefine the association ends of the associations between `Goal`, `Responsibility`, and `Agent`. The latter association ends were not named on the diagram, per UML convention they are `goal`, `responsibility`, and `agent`. Re-interpretation of (Lamsweerde 2009, p.489-490).

The aggregation relationship of the `Agent` meta-class to itself allows for stepwise refinement of an agent model. The requirements analyst starts modeling "abstract agents" to have responsibility for high-level goals. When the abstract `Agent` is decomposed into concrete `Agents`, and the `Goal` into finer `sub-Goals`, each concrete `Agent` is given `Responsibility` for one of the `sub-Goals`.
In our eyes, the term "abstract agent" is confusing as it introduces additional terminology beyond what already is in the meta-model. The question arises whether the meta-class `Agent` should be renamed to `AbstractAgent` or `CompositeAgent`. We would prefer `CompositeAgent`: All elements of a model are abstractions, therefore prefixing the name of any element with the term abstract is a tautology, something that is by definition abstract does not need the term abstract in its name. Also what is most characteristic for the `Agent` meta-class, is in our eyes its decomposability, from which its degree of abstraction immediately results.
Two more observations regarding the decomposability of `Agents`: First, the KAOS meta-model does not provide any facility to indicate where the decomposition ends, therefore we have to assume that all instances of `Agent` are always decomposable. To introduce non-decomposable agents, either constraints on `SoftwareToBeAgent` and `EnvironmentAgent` would need to indicate that these subclasses of `Agent` cannot be part of `Aggregation` associations, a property they do otherwise inherit from the `Object` meta-class. Or the `Object` taxonomy needs to be restructured in a way that the ability to differentiate between decomposable and non-decomposable `Objects` is introduced. Second, the decomposability of `Agents` is defined twice in the KAOS meta-model, once through the `Aggregation` meta-class, explained in the previous subsection, shown in Figure 3.7, and once through an unnamed UML aggregation on the agent meta-model, shown in Figure 3.9. It is questionable whether that aggregation relationship is actually needed.
With `Agents` linked to `Goals` via `Responsibility` links, it might still not be precisely clear which `Agent` instance is responsible for which `Goal` instance. To address this issue, a "responsibility instance declaration", a textual annotation, can be attached to the agent model. That declaration is held in the `instanceVar` attribute of the `Responsibility` meta-relationship.

If an `Agent` is responsible for a certain `Goal`, this means the `Agent` must be capable of realizing that `Goal`. The capability does not automatically follow from an existing model, so it is the obligation of the modeler to show that the capability actually exists. The means for showing that are provided through the `Monitoring` and `Control` meta-relationships. The two relationships link the agent-meta-model with the object meta-model by relating `Agent` with `Association` and `Attribute`. An `Agent` "*monitors an attribute* [...] if its instances can get the values of this attribute" from the `Object` to which the `Attribute` belongs. An `Agent` "*monitors an association* if its instances can evaluate whether this association holds between object instances". An `Agent` correspondingly *"controls an attribute"* if it can set its values. An `Agent` "*controls an association* if its instances can create or delete association instances". We assume that modification of association instances is not mentioned here as it can be expressed in terms of a deletion and creation.

Single Monitoring or Control links can either be about an Association or an Attribute, which is indicated by the xor-constraint in the meta-model. Unfortunately, the agent meta-model does not provide complete cardinalities for the two meta-relationships. From our analysis of Chapter 11, we assume that an Agent instance can have many Monitoring and Control links, which cannot exist independently from Agents. A Monitoring or Control link always connects one Agent with one Association or Attribute. These links are actually not shown as single links on diagrams, but are aggregated into one link connecting an Agent with the Object containing the Attribute or Association.

There is a derived Monitoring or Control relationship not mentioned in the meta-model, but in the text of the book: An Agent is said to monitor or control an Object, it is monitoring or controlling all of its Attributes and Associations. Another kind of Monitoring or Control relationship not mentioned in the meta-model is about monitoring or control of conditions. This is surprising as the monitoring and control of conditions is important for reasoning about Agents and their Responsibilities.

A Goal is *unrealizable* by a specific Agent if the Agent is unable to monitor the Attributes or Associations that are needed to detect whether a certain condition that is specified in a Goal specification holds, or if the Agent is required to predict a future state of these attributes or associations. A Goal is also unrealizable for a specific Agent, if the Agent is unable to control the attribute or association that needs to be constrained as specified in the specification of the Goal. Two additional reasons that are not related to the Agent's capabilities can make a goal unrealizable: if the Goal is specified so that it cannot be achieved, or if a BoundaryCondition exists that makes the Goal unsatisfiable.

Potentially multiple Agents are capable of realizing a specific Goal. To support expressing this, the agent meta-model contains the Assignment meta-relationship. If an Agent is related to a LeafGoal via Assignment, it means that that Agent could become responsible for the LeafGoal. All Agents linked to the LeafGoal form the set from which the responsible Agent (i.e. the Agent that will be linked to the LeafGoal with a Responsibility link) is selected. The sysRef attribute of Assignment can optionally indicate with a textual annotation which of the alternative Assignments is promoted to a Responsibility for which variant of the system under discussion.

In contrast to Responsibility, Assignment can link an EnvironmentAgent to a Requirement (and correspondingly, SoftwareToBeAgent to Expectation). But as EnvironmentAgents can only be made responsible for Expectations, this should not be allowed. To the best of our knowledge, that difference between Responsibility and Assignment is not explained in the KAOS literature. Assignment could be modeled similarly to Responsibility (i.e. as outlined in Fig. 3.13), which would make the agent meta-model more complex, but more precise. Alternatively, a constraint on Assignment would be needed to avoid models where Agent subclasses are potentially responsible for LeafGoal subclasses that they cannot realize by definition.

The notion of agent dependencies has been adapted from i*[5]. In KAOS, a Dependency

[5] See Section 3.2.1, especially Fig. 3.48, how URN, the successor of i* has modeled the Dependency

is a ternary association between two instances of Agent and one instance of Goal, where one of the two Agents is in a Responsibility relationship to that Goal. A Dependency means that one Agent, the *depender*, is dependent on the realization of a Goal, the *dependum*, for which another Agent, the *dependee*, is responsible. The depender might not be able to realize some of the Goals it is responsible for if the dependum is not realized.

The agent meta-model does not specify cardinalities on the associations between Dependency, Agent, and Goal. Figure 3.9 thus reflects our understanding of the matter. As the meta-model does not contain any constraints regarding applicability of the Dependency relationship, the modeler can possibly create invalid models, for example as in Figure 3.14. In the figure, an Agent A1 depends on a high-level Goal G, which is decomposed into a Goal SG for which A is responsible.

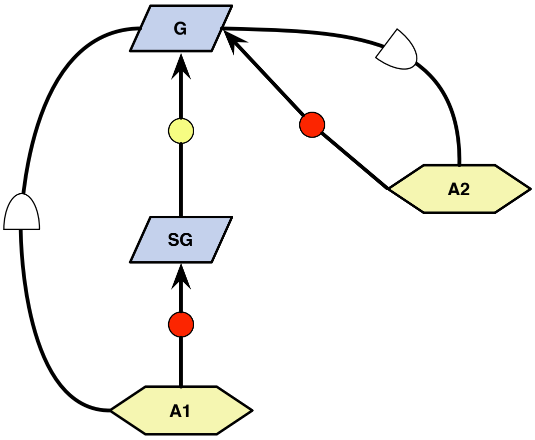


Figure 3.14.: KAOS example with invalid Dependency instance.

3.1.4. Operation meta-model

The central abstraction of the operation meta-model (Fig. 3.15) is the Operation. No other meta-classes are introduced, but various meta-relationships to relate to abstractions of other parts of the KAOS meta-model. The operation meta-model interfaces with the goal meta-model to show the operationalization of goals, with the agent meta-model to show which agents perform which operations, with the object meta-model to show the effect of operations on objects, and with the behavior meta-model to show internal events yielded by an operation.

meta-relationship.

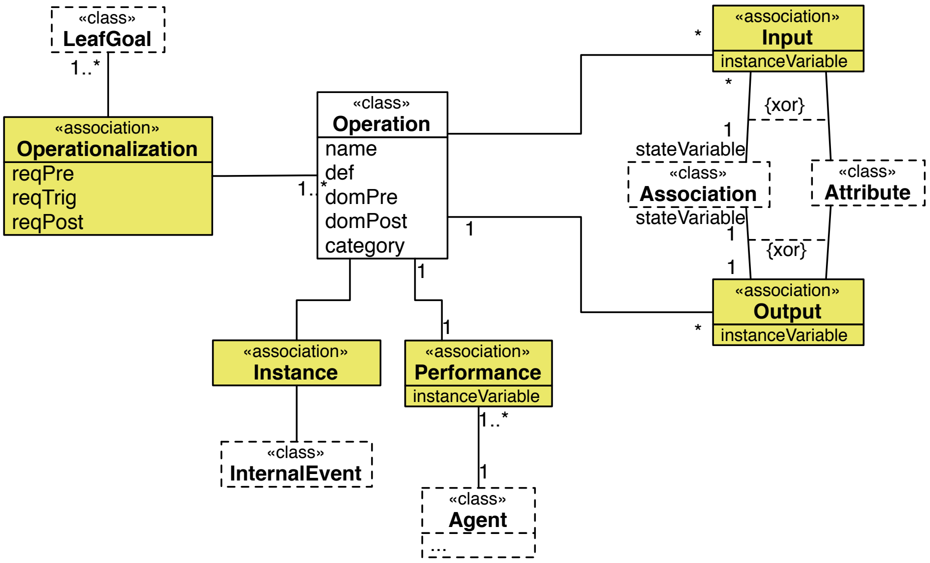


Figure 3.15.: KAOS Operation Meta-Model. Adapted from (Lamsweerde 2009, p.491).

An `Operation` is "a binary relation over system states" (Lamsweerde 2009, p.422). It maps a state before the application of the `Operation` to the state after the application of the `Operation`. The state before the application of the `Operation` is expressed as a tuple of `Associations` or `Attributes`, linked to the `Operation` via instances of the `Input` meta-relationship. In this context, `Objects` owning these `Association` and `Attribute` instances are called *input variables*. The state after the application of the `Operation` is also expressed as such a tuple, linked to the `Operation` via instances of the `Output` meta-relationship. The `Objects` owning these `Association` and `Attribute` instances are then called *output variables*. Input and output variables form the *signature* of the `Operation`. Which `Input` or `Output` link is referring to which `Object`, and how the input or output variable is named in the context of the `Operation`, is tracked by the `instanceVariable` property of the `Input` and `Output` meta-relationships. The `Association` and `Attribute` instances linked by these relationships are called `stateVariables`.
KAOS only considers deterministic `Operations`, i.e. the relation between input and output variables is a function. `Operations` are atomic, i.e. non-decomposable. This is a design decision that favors "goal refinements in a purely declarative model over goal-free operation refinement in an operational model", reflecting the conceptual dominance of goals in KAOS. The argument is that the decomposition of operations is often arbitrary, and handling goal decompositions along operation decompositions in the same model makes satisfaction arguments more complex as needed. With decomposable operations, the requirements model certainly becomes more complex. However, whether operation decomposition seems arbitrary depends on whether the rationale for the decomposition is explicit in the model or not. In the same fashion as KAOS introduces tractable criteria for decomposing goals, a language with different focus could provide such criteria

for decomposing operations.

Operations have a unique name and a natural language definition (def). The state of the input variables before the execution of the Operation is specified declaratively by the domPre property. Analogously, the domPost property specifies the state of the output variables after the execution of the Operation. The two properties rely upon the instanceVariable property of the Input and Output meta-relationships for that specification. An optional attribute category can be used to annotate the Operation to indicate whether it is performed by a SoftwareToBeAgent or an EnvironmentAgent. It can be derived from the Agent instance that is connected to the Operation via the Performance relationship, which maps exactly one Agent to the Operation. The linked Agent must be consistent with its capabilities defined in the agent model. That means that the Associations/Attributes of the Operation's input variables must be monitored by the Agent, and the Associations/Attributes of its output variables must be controlled by the Agent.

The Operationalization meta-relationship links the Operation to a LeafGoal. It is an important part of the argument whether the system under discussion correctly addresses its Goals. With the previously discussed views of a KAOS model, the modeler can show what an Agent is capable of in terms of Objects, and what Goals the Agent is responsible for. With the operation view, the modeler can show which Agent actually performs an Operation, and with the Operationalization link, argue that the Operation actually ensures the satisfaction of a Goal. It follows implicitly from the constraint on the Performance link, stating that only an Agent with the appropriate capabilities can perform an Operation, that the linked LeafGoal is an instance of Requirement if the Agent is an instance of SoftwareToBeAgent, and an instance of Expectation respectively, if the Agent is an instance of EnvironmentAgent.

Operations can satisfy multiple LeafGoals. For each Operationalization link, three conditions can potentially be modeled. The reqPre property "captures a *permission*", it states that the Operation may be performed under the specified condition. The reqTrig property "captures an *obligation*", it states that the Operation must be performed under the specified condition. The reqPost property captures an effect, it states the condition that must be true after the application of the Operation.

The Instance meta-relationship, linking Operation to InternalEvent is only tersely mentioned in the description of the operation meta-model, but in more detail in the description of the behavior meta-model. We therefore assume it should actually be part of the behavior meta-model.

3.1.5. Behavior meta-model

The two central abstractions of the behavior meta-model (Fig. 3.16) are AgentSM and Scenario. AgentSM models the behavior of all instances of an Agent, a Scenario models the interaction between specific instances of various Agents. Both abstractions refer to a Goal. A Scenario is related to Goal by the InstanceCoverage meta-relationship. It expresses that the Scenario constitutes one particular behavior that addresses the Goal. An AgentSM is connected to Goal via the ClassCoverage meta-relationship. It expresses that the AgentSM captures any behavior that addresses the covered Goal. Bridging the state machine and scenario parts of the behavior meta-model, the Coverage meta-relationship links the AgentSM to all Scenarios in which the instances of the Agent, whose behavior the AgentSM constitutes, participate. In the following paragraphs we first describe the partition of the behavior meta-model focusing on Scenario, and then the partition focusing on AgentSM.

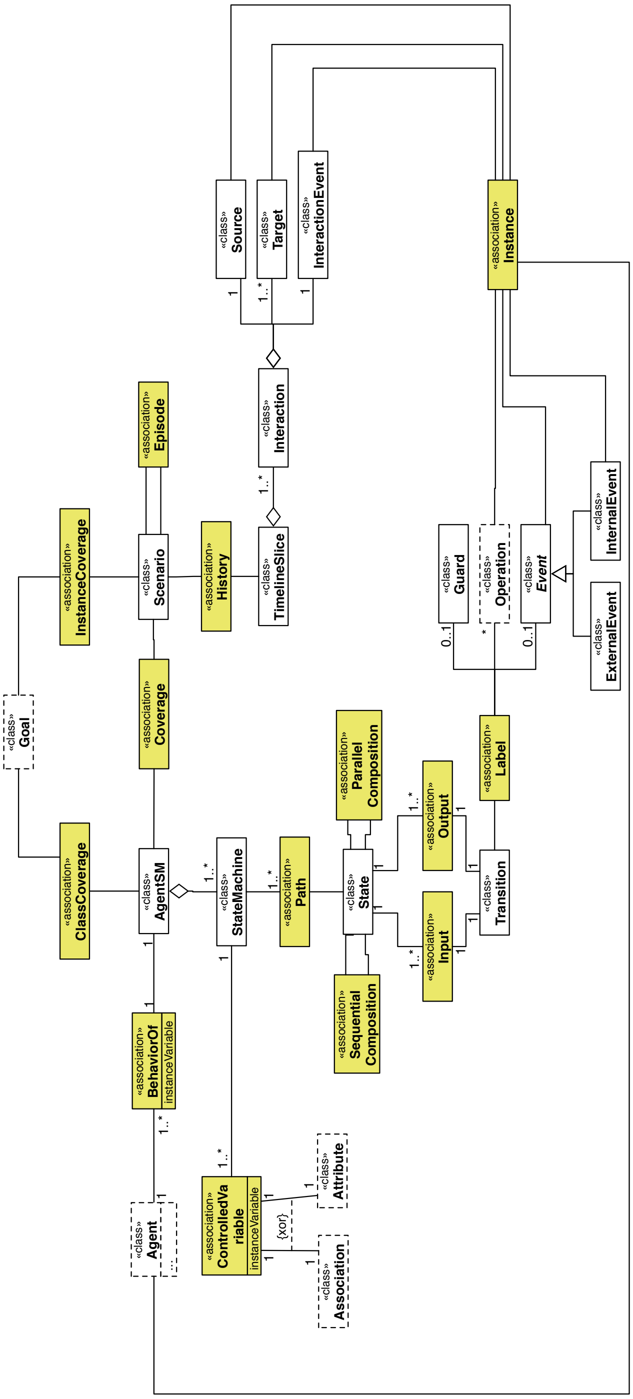


Figure 3.16.: KAOS Behavior Meta-Model. Adapted from (Lamsweerde 2009, p.492).

`Scenarios` are further classified into *positive* and *negative* `Scenarios`, which is not visible in the meta-model. A positive Scenario captures the "admissible behavior" of the interacting Agent instances, whereas a negative Scenario captures "inadmissible behavior", i.e. it captures a particular behavior satisfying an Obstacle to the covered Goal. To the best of our knowledge, the reason why the distinction between positive and negative scenarios is not explicit in the meta-model is not presented in the book. Scenarios can also be *optional*, which is also not reflected by the meta-model.

The Episode meta-relationship allows for a sequential decomposition of Scenarios into sub-Scenarios, for the purpose of complexity management on diagrams. The sub-Scenarios should cover sub-Goals of the Goal covered by the parent Scenario, but the meta-model contains no constraint to enforce this rule.

A Scenario is composed of TimelineSlices, linked to the Scenario via the History meta-relationship. The TimelineSlices are sequentially ordered. A TimelineSlice in turn consists of one to any Interactions that happen in parallel. An Interaction is constituted of an InteractionEvent, that takes place between a Source and a Target, both being Agent instances. The InteractionEvent is an instance of Event.

The Instance meta-relationship is confusing in our eyes. As it is presented in the meta-model, it can either link Source to Agent, or Target to Agent, or InteractionEvent to Event, or InternalEvent to Operation (Fig. 3.17). Without accompanying constraints it is impossible to implement this meta-relationship correctly. We think it should be split into dedicated relationships. Furthermore, the meta-classes Source, Target, and InteractionEvent can be modeled as roles of the classes contained by the Interaction. It is actually surprising that the behavior meta-model has no abstractions for Agent and Event instances, as both of them have a notational representation on KAOS sequence diagrams. Figure 3.18 shows a refactored meta-model that we propose. It is more precise regarding Agent and Event instances to be represented on KAOS diagrams.

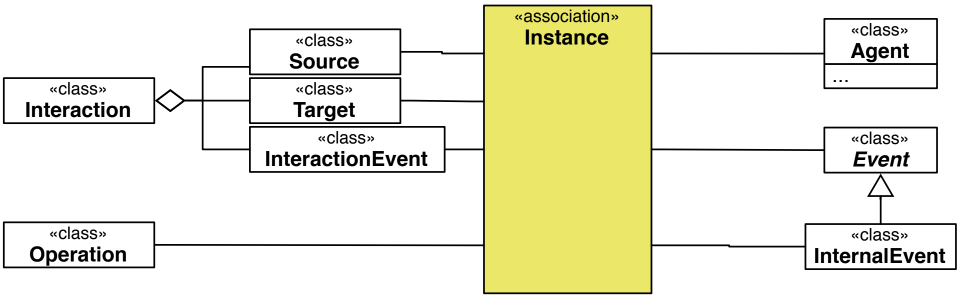


Figure 3.17.: KAOS Behavior Meta-Model excerpt with focus on the Instance meta-relationship. Adapted from (Lamsweerde 2009, p.491f).

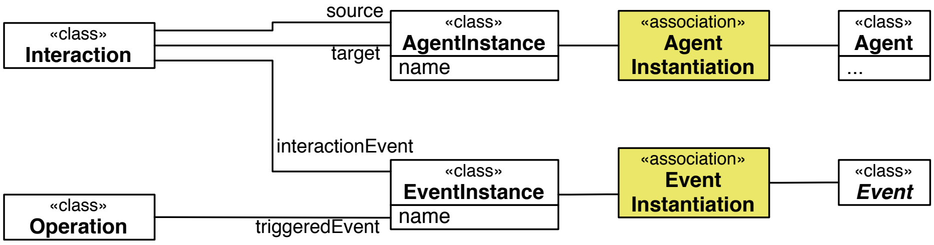


Figure 3.18.: Alternative model in which it is more precise what meta-classes `Interaction` and `Operation` are interacting with.

The oddly named `AgentSM` constitutes the behavior of an `Agent`, which is expressed by the `BehaviorOf` meta-relationship. The “SM” in `AgentSM` stands for state machine, which is at first sight irritating as the `AgentSM` consists of `StateMachines`. A simpler yet intuitive name for `AgentSM` could have been Behavior. A `StateMachine` constitutes only the behavior of the `Agent` dealing with one controlled variable, which is expressed by the `ControlledVariable` meta-relationship. Analogous to the `Input`, `Output`, `Monitor`, and `Control` meta-relationships from the other parts of the meta-model, a controlled variable is either an `Association` or an `Attribute` of an `Object`. The purpose of the `instanceVariable` property of `BehaviorOf` is unclear to us, as the `AgentSM` covers the behavior of all instances of the `Agent` - therefore it is unclear for which purpose single `Agent` instances should be identified in the context of `BehaviorOf` links.
A `StateMachine` consists of a number of potentially composite `States`, building a path. This is expressed by the `Path` meta-relationship. If a `State` is composite, its substates are either in sequential order or are parallel to each other. This is expressed by the `Sequential-` and `ParallelDecomposition` meta-relationships. The meta-model is lacking a constraint here, or a composite pattern, to avoid that a `State` may include itself, or that a `State` includes one of its ancestor states. There are special kinds of `State` called initial state, final state, initial sub-state, and final sub-state, that do not inherit the two composition relationships, but these are not mentioned in the meta-model “for the sake of clarity of the diagram”. We are not convinced that this incompleteness of the meta-model facilitates a good understanding of the behavior meta-model as important parts of the semantics of `State` are missing. Consider Figure 3.19 for a sketch of an alternative model.

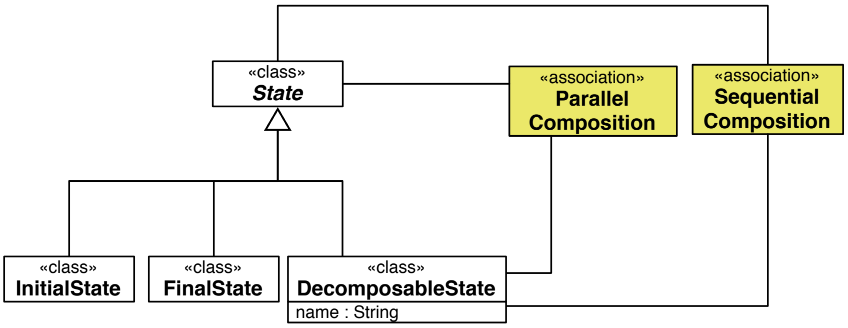


Figure 3.19.: Alternative model in which the `State` taxonomy is explicated.

The name of the `Path` meta-relationship is misleading as it only links `States` to the corresponding `StateMachine`. In our eyes, the actual path is not only constituted by the `States`, but also the `Transitions` between them. A `Transition` maps an input to an output `State`, which is which is expressed by the `Input` and `Output` meta-relationships[6] that link `States` with `Transitions`. `Transitions` can happen automatically, which is often the case with `Transitions` outgoing from the initial `State` of a `StateMachine`. More often, they are bound to a certain `Event`. When the `Event` occurs, the `Transition` takes place. If the `Transition` can only occur if the `Event` takes place and a certain condition holds, the `Transition` is additionally linked to a `Guard`. `Transitions` can cause auxiliary `Operations` to be executed. These `Operations` are also called *actions* in the book. If all incoming `Transitions` of a `State` (i.e. all `Transitions` in which the `State` is linked via the `Output` relationship) are associated with the same auxiliary `Operation`, that `Operation` can be understood as an *entry action* of the state. Analogously, if all outgoing `Transitions` of a `State` have the same auxiliary `Operation`, this `Operation` can be understood as an *exit action* of the `State`. Entry and exit actions modify the notation of the `State`, therefore we suggest that derived properties should be added to `State`, computing entry or exit actions from the existing `Input` and `Output` relationships. Another kind of auxiliary `Operation` is the *event notification*, which seems to be syntactic mechanism for complexity management. It can be used on diagrams to express that a `Transition` in one `StateMachine` leads to an `Event` in another `StateMachine`. Under which circumstances the `Operation` linked to the `Transition` is an event notification is not visible in the meta-model. All these properties of a `Transition` are linked to the `Transition` via the `Label` meta-relationship, defining that a `Transition` can optionally have a `Guard`, zero to many auxiliary `Operations`, and an optional `Event`. We think the name `Label` is misleading as it implies a notational purpose of the meta-relationship. However, the `Guard`, `Event`, and `Operation` instances linked to the `Transition` constitutes a major part of its semantics, they are not just labeling the `Transition`, they are a major part of its definition. In the original meta-model, the `Label` relationship branches out into three branches connected by a dashed line, which

[6] We assume that the behavior meta-model has its own namespace, otherwise these meta-relationships would have a name conflict with the equally named meta-relationships of the operation meta-model.

should be interpreted as an OR-association. We have interpreted this in terms of the cardinalities on the associations connecting `Label` with `Guard`, `Event` and `Operation`. `Events` are further classified into `ExternalEvents` and `InternalEvents`. `ExternalEvents` are `Events` that are not controlled by the `Agent` associated with the `StateMachine`. `InternalEvents` in turn are controlled by that `Agent`. An `InternalEvent` corresponds to the application of an `Operation` by an Agent. This is expressed by the `Instance` relationship mentioned above. In this context, it means that the applied `Operation` yields an instance of `InternalEvent`. This way, the modeler could, for a given `Operation`, check how the `Operation` is participating in the general behavior of the `Agent`. However, there is no notational counterpart for this relationship, but a convention that suggests "choosing a suggestive verb for an operation name and the corresponding noun for the internal event". The example given in the book mentions a `CloseDoors` `Operation` and a `doorsClosing` `Event`. We are not convinced that this will work in a complex model, even more so when assuming that a convention is not necessarily followed by every modeler. It would be more straightforward to have a dedicated diagram showing which `Operation` yields which `InternalEvents`.

3.1.6. Example Model

The example model presented in this section is dedicated to describing the requirements of an example problem called the Barbados Car Crash Crisis Management System. The sole source of information regarding this system is the bCMS requirements document (Capozucca et al. 2012) (from here on simply called 'requirements document'). The model presented in this section would therefore reflect a state that the requirements analyst has prepared from that initial requirements document, and which will afterwards be elaborated via dedicated meetings, phone calls, or comparable communication. The requirements document is not about a single system under discussion, but about a software product line. Therefore multiple variation points have to be taken into account. The purpose of any system variant is to support the coordination between a fire and a police station to resolve a "car crash crisis". The example scenario of the requirements document is "an accident involving an overturned oil tanker on a highway, where the tanker is on fire". The scope of the system is to support the coordination between stations, each of which has a dedicated coordinator role. The task of the coordinators is to plan how police and fire vehicles get to the crisis location as quickly as possible, and to decide about the optimal number of vehicles. Out of the scope of the system is the reception of witness reports, and the actual execution of the missions of the fire and police men.

While the problem might seem oversimplified, this simplification has deliberately been made by the creators of the example problem to achieve a better comparability of models in different languages. As the problem will also be presented in terms of a URN and a URML model in this dissertation, this was an argument for choosing that concrete example problem. The example model is based on our own judgement to gain greater flexibility regarding the choice which kinds of diagrams to include and which not. A comparable model of the example problem expressed in KAOS has been done by Cailliau et al. (Cailliau et al. 2013a, 2013b).

The goal graph is described in top-down fashion, which does not mean it has been analyzed top-down. It is just the mode of presentation. The presentation starts with the top-level goals, each of which is decomposed into finer goals, until the presentation ends at the finest goals that can be assigned to a single agent. Then an object model is presented. The object model allows to present the capabilities of the agents. The capabilities are expressed in terms of links of agents to entities of the object model. Then some operations performed by the modeled agents are described and are related to the object and goal model.
The primary goal to be addressed by the bCMS is to resolve the crisis quickly and cost-effectively. As the goal is about achieving a desired effect (resolution of crisis) in the near future, it is modeled as an achieve goal in Figure 3.20. Some of the goal's features are presented in model annotation attached to the goal. For example, the annotation provides a hint from which source the goal was externalized. The goal is further AND-decomposed into a domain hypothesis and two more achieve goals. The hypothesis is that the existence of a crisis is already known at the police and fire stations, so that the need for crisis resolution is already present. What is to be achieved to resolve the crisis, is that a communication channel is provided, and that a coordination process is performed.

The nature of the communication channel between the two coordinators is unclear: Do they only interact indirectly via the user interface of bCMS or should bCMS offer a possibility for textual communication or a voice-based communication? Or can it be assumed that the coordinators can additionally rely upon a phone line or some audio-visual communication channel like a videoconferencing system.

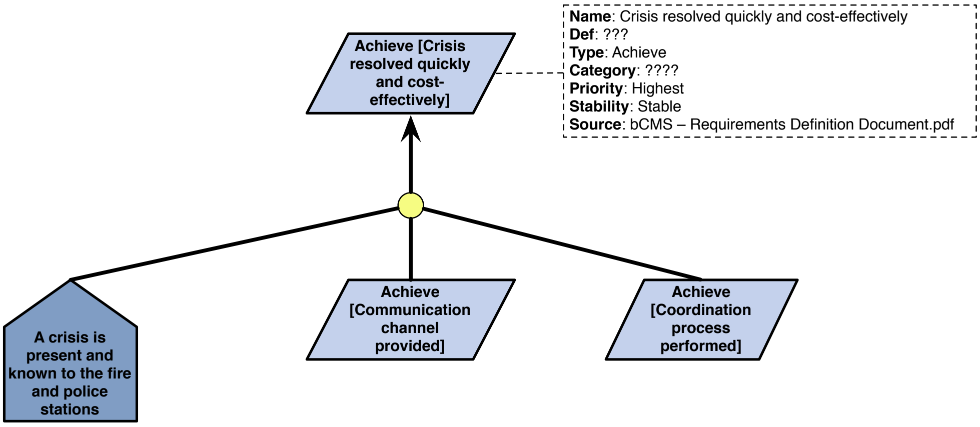

Achieve [Crisis resolved quickly and cost-effectively]
Name: Crisis resolved quickly and cost-effectively
Def: ???
Type: Achieve
Category: ????
Priority: Highest
Stability: Stable
Source: bCMS – Requirements Definition Document.pdf
A crisis is present and known to the fire and police stations
Achieve [Communication channel provided]
Achieve [Coordination process performed]

Figure 3.20.: Resolution of crisis involves provision of a communication channel and a successful coordination process.

For the communication channel to be provided, we assume that the system to be developed will run on a hardware platform that includes a wired connection via a T1 link and on a software platform that supports the HTTPS protocol for secure communication

(Fig. 3.21). The objective for bCMS is to establish a connection over HTTPS to connect the police and the fire station. In addition, there is the expectation that *somebody* will take care of ensuring that connection hardware works reliably, e.g. some network administrator that will periodically ensure routers, uplinks, and cables are working. As no information about this is provided in the bCMS requirements document, that environment agent has no name yet. In addition, a model annotation is attached to describe the issue.

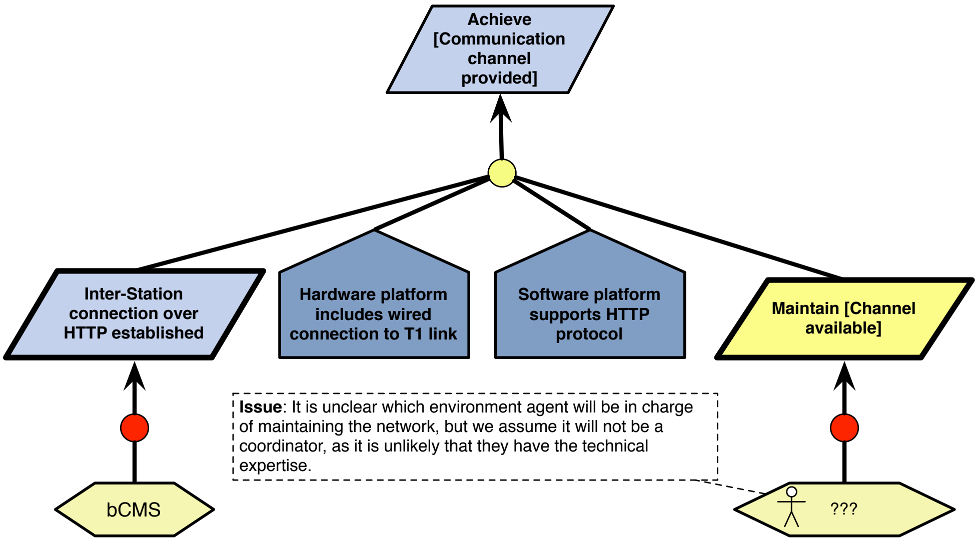

Achieve [Communication channel provided]
Inter-Station connection over HTTP established
Hardware platform includes wired connection to T1 link
Software platform supports HTTP protocol
Maintain [Channel available]
Issue: It is unclear which environment agent will be in charge of maintaining the network, but we assume it will not be a coordinator, as it is unlikely that they have the technical expertise.
bCMS
???

Figure 3.21.: Software agent bCMS and an not-yet-defined environment agent

Figure 3.22 shows the decomposition of the goal Coordination process performed. It can be broken down into the achievement of the initiation of a coordination session, the exchange of crisis details, and the coordination of the route plan.

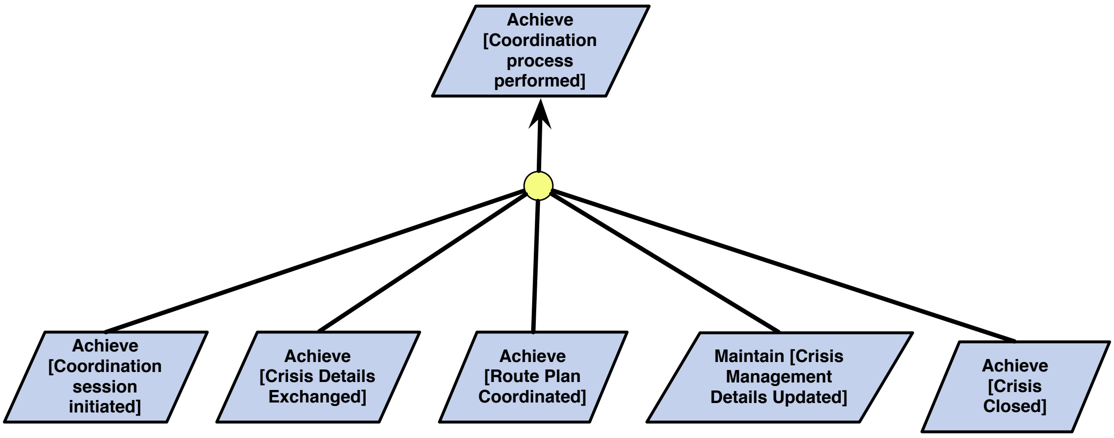

Achieve [Coordination process performed]
Achieve [Coordination session initiated]
Achieve [Crisis Details Exchanged]
Achieve [Route Plan Coordinated]
Maintain [Crisis Management Details Updated]
Achieve [Crisis Closed]

Figure 3.22.: Coordination process performed

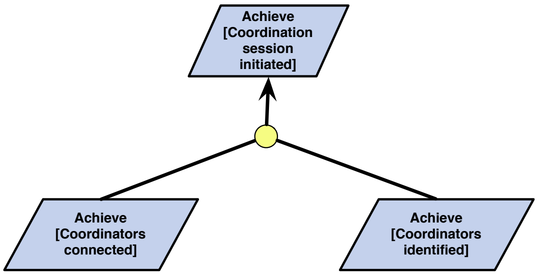

Achieve [Coordination session initiated]
Achieve [Coordinators connected]
Achieve [Coordinators identified]

Figure 3.23.: Coordination session initiated

Figure 3.23 shows that Coordination session initiated can be further broken down into Coordinators connected and Coordinators identified. Both goals are still no requirement for bCMS as both can only be achieved through the cooperation of multiple agents.

Coordinators connected can be decomposed one more time, as shown in Figure 3.24. Session created is a requirement for which the bCMS software-to-be agent is responsible. The other two subgoals are expectations on the environment agents PSC and FSC, stating that the two agents are expected to connect soon after they become aware of a crisis to address.

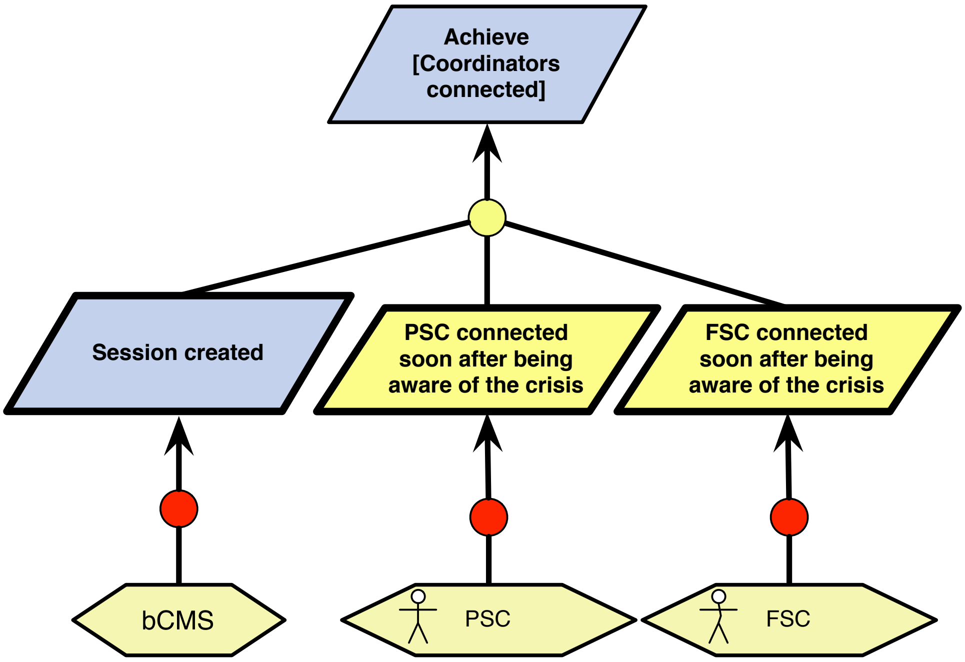

Achieve [Coordinators connected]
Session created
PSC connected soon after being aware of the crisis
FSC connected soon after being aware of the crisis
bCMS
PSC
FSC

Figure 3.24.: Coordinators connected can be decomposed into one requirement and two expectations.

In one possible decomposition of Coordinators identified (Figure 3.25), bCMS is required to provide an authentication form, and each of the two coordinators is expected to fill the respective form. bCMS is then responsible for achieving Username and Password Verified. As an alternative, the identification of coordinators could be achieved via a certificate-based technique. That decomposition is shown in Figure 3.26. Here it is assumed that the bCMS can use facilities of the platform it is running on to access file storage. Identification is then achieved if a certificate has been requested, both coordinators provided their respective certificate, and both certificates have been successfully verified. Showing the two alternative decompositions of `Coordinators identified` in one diagram seems sensible, the alternative refinements can be given a name to facilitate discussions about the alternatives. However, such diagrams quickly gain complexity. The bCMS requirement document actually discusses five approaches to identification, which are not even strictly alternative but should be combinable. As Figure 3.27 suggests, where already the presentation of two alternative decompositions leads to considerable complexity, it is not possible to put the combined requirements knowledge into a digestible KAOS diagram. KAOS lacks complexity management mechanisms here. When splitting up the knowledge to multiple diagrams, inter-diagram links are missing or any indications that not the complete existing knowledge is shown on the diagram. When putting everything into one diagram, the diagram becomes unreadable.

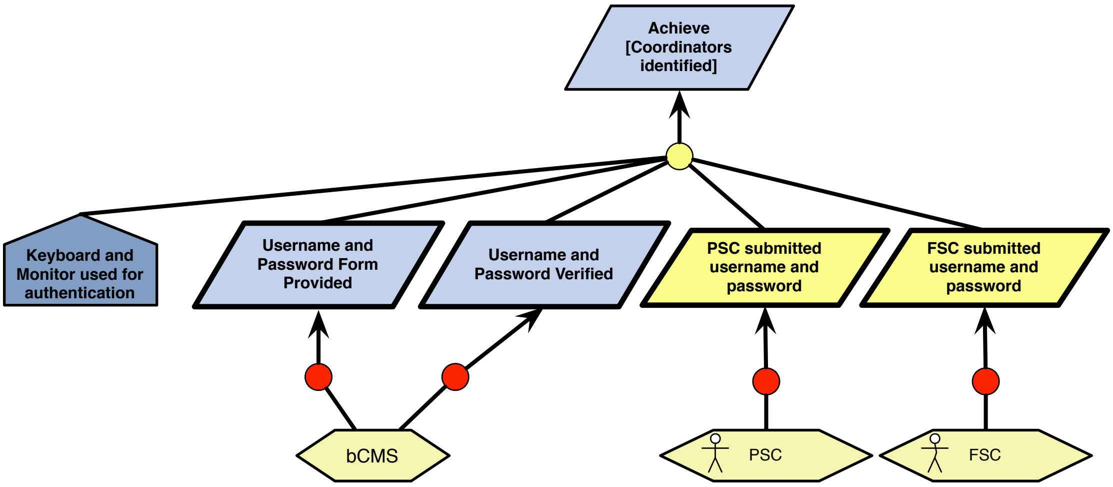

Achieve [Coordinators identified]
Keyboard and Monitor used for authentication
Username and Password Form Provided
Username and Password Verified
PSC submitted username and password
FSC submitted username and password
bCMS
PSC
FSC

Figure 3.25.: Password-based alternative to decompose `Coordinators identified`

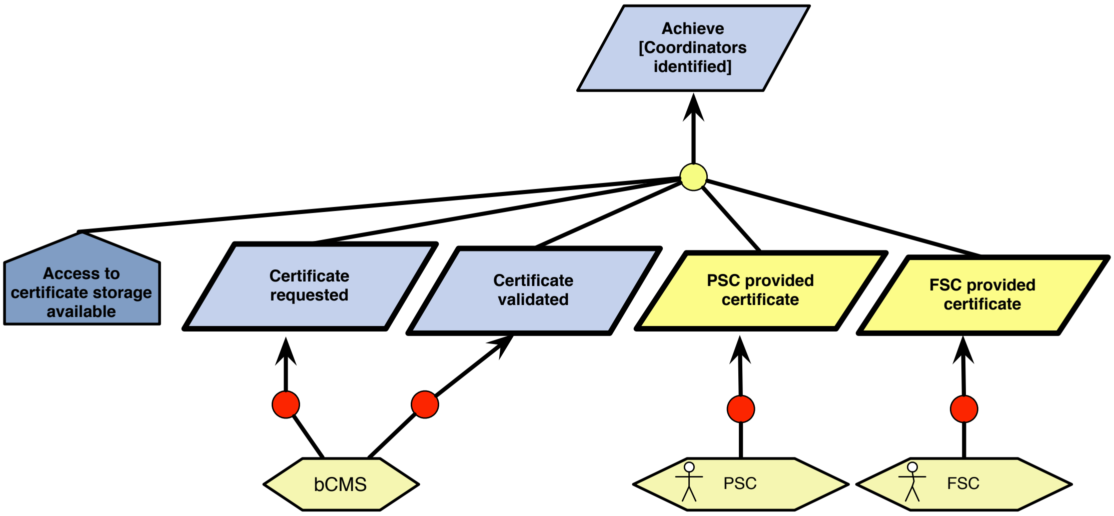


Figure 3.26.: Certificate-based alternative to decompose `Coordinators identified`

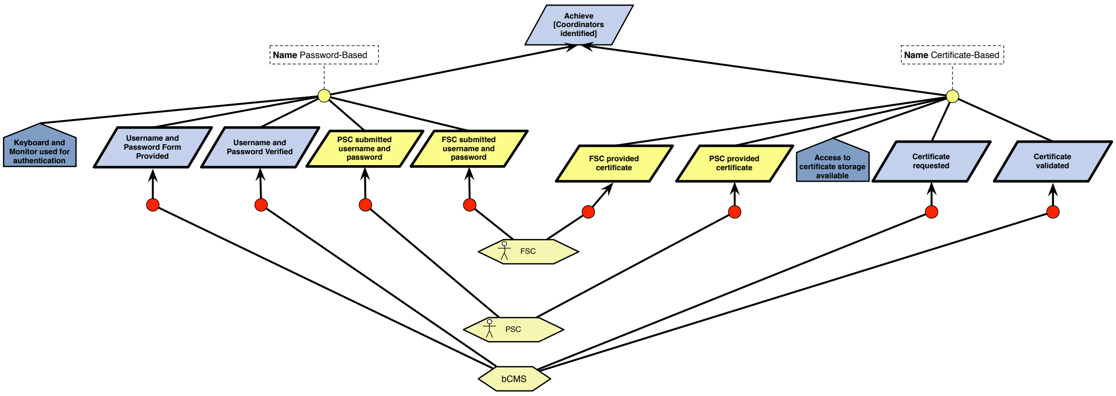

Achieve [Coordinators identified]
Name Password-Based
Name Certificate-Based
Keyboard and Monitor used for authentication
Username and Password Form Provided
Username and Password Verified
PSC submitted username and password
FSC submitted username and password
FSC provided certificate
PSC provided certificate
Access to certificate storage available
Certificate requested
Certificate validated
FSC
PSC
bCMS

Figure 3.27.: Two alternative decompositions of `Coordinators identified`, annotated with refinement names.

For `Crisis Details Exchanged`, the system is required to achieve that the knowledge of the coordinators can be represented, and given that both coordinators have provided their knowledge, that knowledge is merged into one representation (Figure 3.28). From the source document it is not exactly clear what the purpose of the exchange of crisis details is. Therefore a model annotation describing the issue is attached, in which the assumption of the current model is elaborated. It might turn out that there is a need for resolution of conflicting knowledge about the crisis. In that case, an iteration of the model excerpt would entail another requirement. What the crisis knowledge actually entails, is shown in the object model of Figure 3.33.

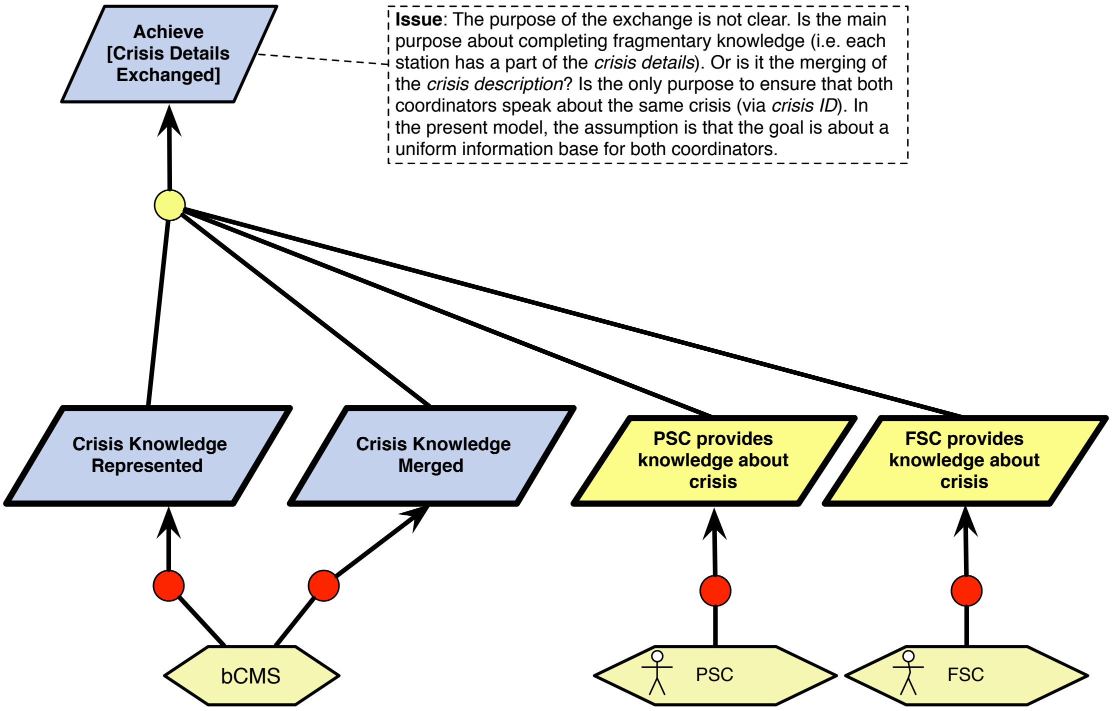

Achieve
[Crisis Details
Exchanged]
Issue: The purpose of the exchange is not clear. Is the main purpose about completing fragmentary knowledge (i.e. each station has a part of the *crisis details*). Or is it the merging of the *crisis description*? Is the only purpose to ensure that both coordinators speak about the same crisis (via *crisis ID*). In the present model, the assumption is that the goal is about a uniform information base for both coordinators.
Crisis Knowledge Represented
Crisis Knowledge Merged
PSC provides knowledge about crisis
FSC provides knowledge about crisis
bCMS
PSC
FSC

Figure 3.28.: Requirements and expectations regarding `Crisis Details Exchanged`

The next subgoal of `Coordination process performed`, `Route Plan Coordinated`, is decomposed into three subgoals (Figure 3.29). Each of the three subgoals involves multiple agents, therefore no responsibility can be modeled yet. Responsibilities can be modeled one level deeper in the goal hierarchy, as shown in Figures 3.30, 3.31, and 3.32.

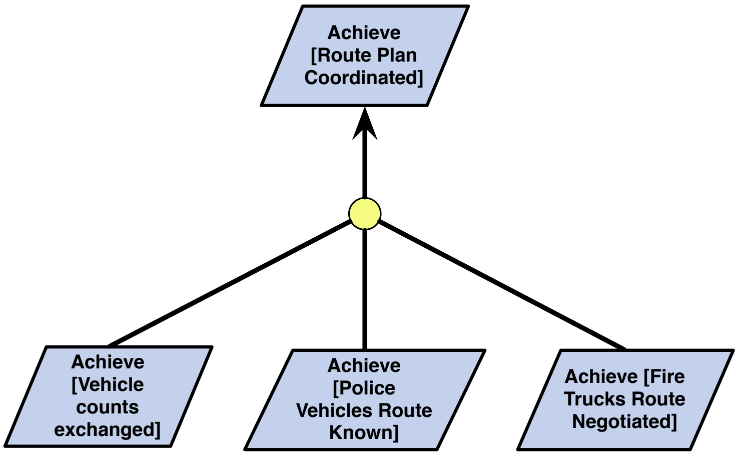


Figure 3.29.: Refinement of `Route Plan Coordinated`

To achieve `Vehicle counts exchanged`, it is expected that both coordinators provide the respective number of necessary vehicles. bCMS is responsible for achieving the recording of the vehicle counts.

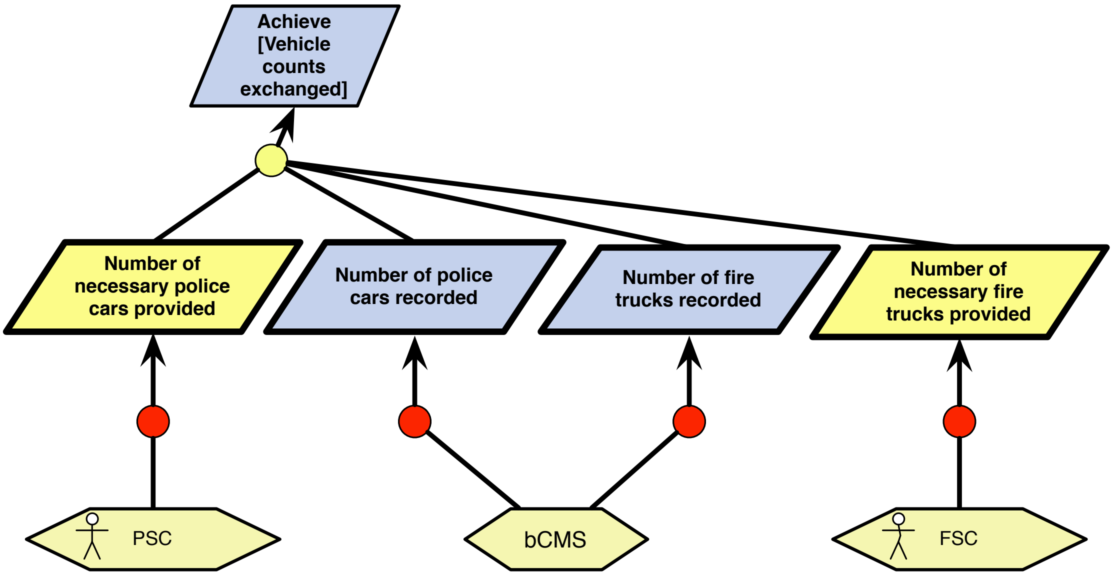

Achieve [Vehicle counts exchanged]
Number of necessary police cars provided
Number of police cars recorded
Number of fire trucks recorded
Number of necessary fire trucks provided
PSC
bCMS
FSC

Figure 3.30.: Three agents are involved in achieving Vehicle counts exchanged.

For Police Vehicles Route Known it is expected that the PSC provides the route (Figure 3.31). bCMS is required to record the route and present it to both coordinators. FSC is expected not to ignore the route, but from the requirements document it could not be derived that the FSC has any influence on that route.

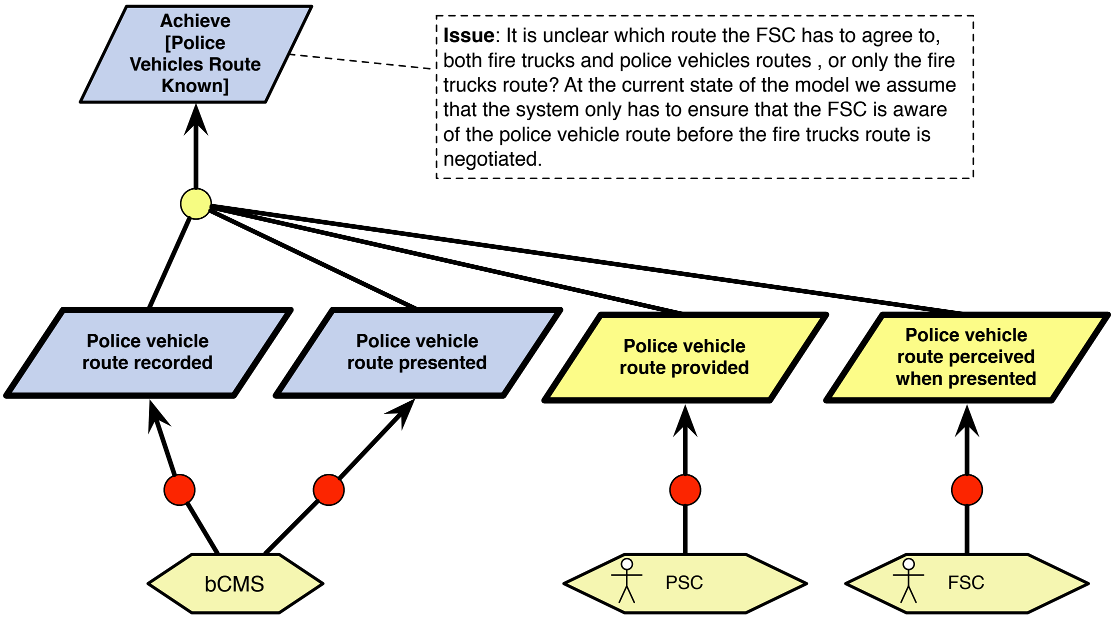

Achieve [Police Vehicles Route Known]
Issue: It is unclear which route the FSC has to agree to, both fire trucks and police vehicles routes , or only the fire trucks route? At the current state of the model we assume that the system only has to ensure that the FSC is aware of the police vehicle route before the fire trucks route is negotiated.
Police vehicle route recorded
Police vehicle route presented
Police vehicle route provided
Police vehicle route perceived when presented
bCMS
PSC
FSC

Figure 3.31.: `Police Vehicles Route Known`

The goal graph for `Fire Trucks Route Negotiated` is much more complex as the `FSC` can veto the route proposed by the `PSC`. A negotiation of the fire truck route entails that the route is recorded when confirmed, and iterated when discarded by the `FSC`. Furthermore, it entails cancellation of the negotiation by the `PSC` if no more route is left to propose. The iteration subgoal is recursive as for the new proposed route the same negotiation is to be achieved.

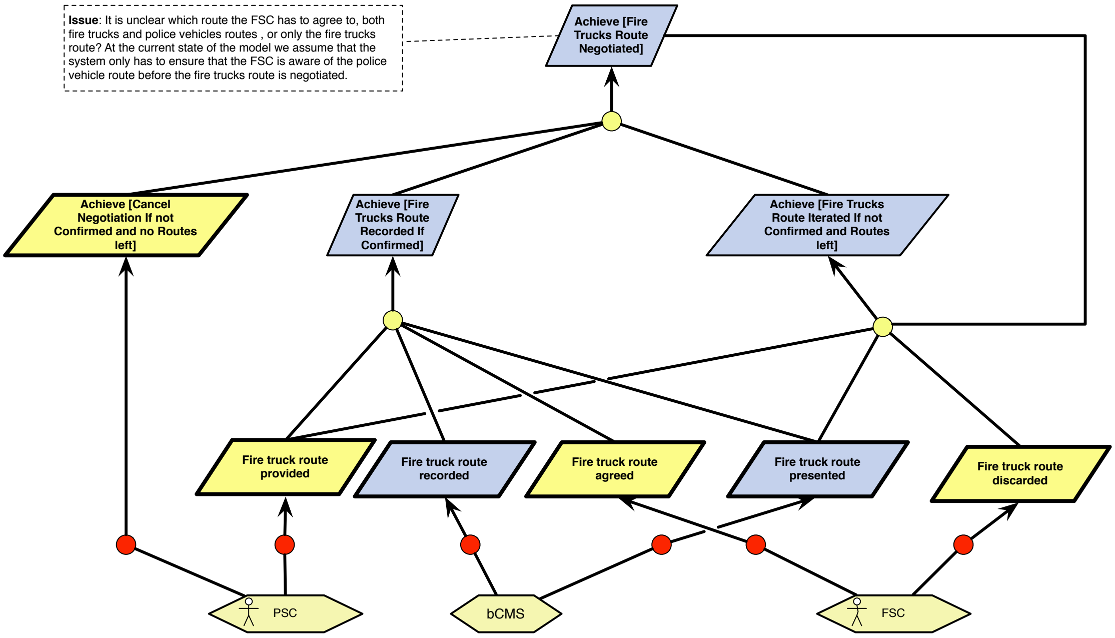

Issue: It is unclear which route the FSC has to agree to, both fire trucks and police vehicles routes , or only the fire trucks route? At the current state of the model we assume that the system only has to ensure that the FSC is aware of the police vehicle route before the fire trucks route is negotiated.
Achieve [Fire Trucks Route Negotiated]
Achieve [Cancel Negotiation If not Confirmed and no Routes left]
Achieve [Fire Trucks Route Recorded If Confirmed]
Achieve [Fire Trucks Route Iterated If not Confirmed and Routes left]
Fire truck route provided
Fire truck route recorded
Fire truck route agreed
Fire truck route presented
Fire truck route discarded
PSC
bCMS
FSC

Figure 3.32.: `Fire Trucks Route Negotiated` is decomposed recursively

The requirements document contains a section titled "Data Dictionary" that contains clear indications which information must be tracked by `bCMS`. From the description of the coordination process additional details were derived. For some attributes types were not deducible from the requirements document. In that case, as for the `Identifier` attribute of `CrisisDetails`, the type was left unspecified. The resulting class diagram is shown in Figure 3.33. The notation is similar to UML, but has some peculiarities - for example, the convention to use uppercase names for attributes, the declaration of enumerations, or the placement of multiplicities within attribute declarations. For each crisis to be dealt with there should be exactly one instance of `CoordinationSession` and exactly one related instance of `CrisisDetails`. `CoordinationSession` has not been explicitly mentioned in the requirements document, but a place to store the `StartTime` of the session is needed, otherwise `bCMS` would not be capable of creating `TimeoutLog` instances. These are created if the coordination is taking longer than some pre-defined `TimeoutInterval` - which also should be stored somewhere. `TimeoutInterval` is not expected to change over the runtime of a session. In KAOS this is called a rigid attribute, indicated by the # sign in the attribute declaration. When an instance of `TimeoutLog` is created, `bCMS` provides values for `Time` and `Date`, but has to ask each coordinator to specify a reason. Any further purpose of `TimeoutLog` has not been discussed in the requirements document. It could be used for some a posteriori analysis of timeout reasons, which could potentially be used to improve the bCMS software in some unspecified future.

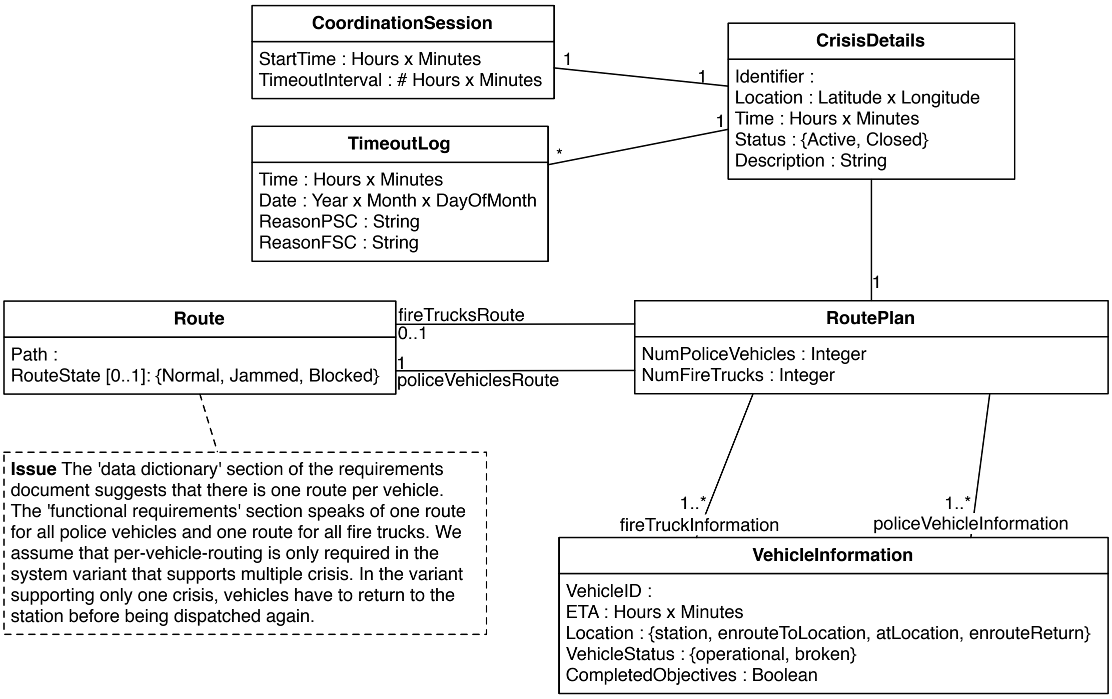
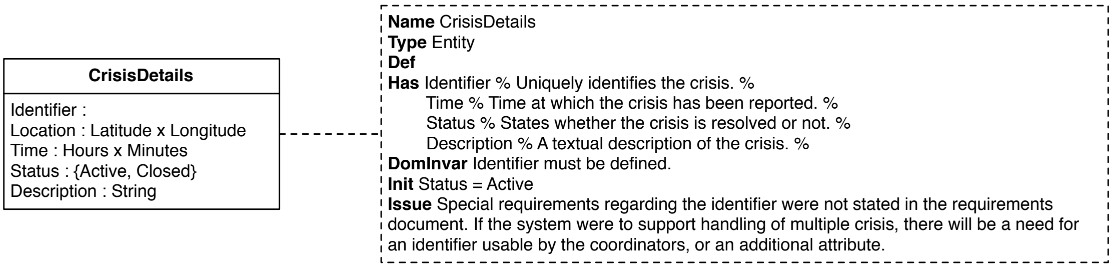

CoordinationSession
StartTime : Hours x Minutes
TimeoutInterval : # Hours x Minutes
1
1
CrisisDetails
Identifier :
Location : Latitude x Longitude
Time : Hours x Minutes
Status : {Active, Closed}
Description : String
1
*
TimeoutLog
Time : Hours x Minutes
Date : Year x Month x DayOfMonth
ReasonPSC : String
ReasonFSC : String
1
Route
Path :
RouteState [0..1]: {Normal, Jammed, Blocked}
fireTrucksRoute
0..1
1
policeVehiclesRoute
RoutePlan
NumPoliceVehicles : Integer
NumFireTrucks : Integer
Issue The 'data dictionary' section of the requirements document suggests that there is one route per vehicle. The 'functional requirements' section speaks of one route for all police vehicles and one route for all fire trucks. We assume that per-vehicle-routing is only required in the system variant that supports multiple crisis. In the variant supporting only one crisis, vehicles have to return to the station before being dispatched again.
1..*
fireTruckInformation
1..*
policeVehicleInformation
VehicleInformation
VehicleID :
ETA : Hours x Minutes
Location : {station, enrouteToLocation, atLocation, enrouteReturn}
VehicleStatus : {operational, broken}
CompletedObjectives : Boolean

Figure 3.33.: Software Objects of bCMS

The `CrisisDetails` instance holds an `Identifier` attribute to uniquely identify the crisis to be resolved. A `Location` attribute holds a GPS coordinate to specify where the crisis happened, and a `Time` attribute captures when the crisis happened. The `Status` attribute indicates whether the crisis has been closed or not. `Description` is a free-text attribute to add additional details, potentially gathered from witness reports - the sources of that information however are out of the scope of this model. It is assumed that `PSC` and `FSC` provide the respective attribute values in the process step where crisis details are exchanged. The main data of the coordination session are captured in the `RoutePlan` instance. It holds the number of necessary police vehicles and fire trucks, in `NumPoliceVehicles` and `NumFireTrucks`. It further refers to the planned route for police vehicles and the planned route for fire trucks. There may be no planned route for fire trucks if no agreement could be reached between `PSC` and `FSC`. The `Route` class specifies the path for the vehicles. The type of the `Path` attribute has yet to be elicited, as there was no clear indication about its structure in the requirements document. From the process description, the `RouteState` attribute was derived. If the value of the attribute is changed to `Jammed` or `Blocked`, `PSC` or `FSC` have to decide about sending replacement vehicles and also have to notify each other about changed estimated time of arrival (`ETA`) of the vehicles on that route. The `ETA` is an attribute of the the `VehicleInformation` class. It has a `VehicleID` to uniquely identify the physical vehicle this information is about. `Location` does not provide a real-time location but a discretization to report the vehicles state in view of the coordination process: A vehicle location is specified as being at the station, en route to the crisis location, at the crisis location, or en route back to the station. `VehicleStatus` is again an attribute not literally specified in the "Data Dictionary". It was derived from the description of the coordination process, as the coordinators have to decide about replacement vehicles and report a new ETA to the other coordinator, if a vehicle broke down on its way to the crisis location. `CompletedObjectives` is a boolean attribute to indicate that a vehicle is done and can return to the station. It is needed in addition to the `enrouteReturn` value of `Location`, as a vehicle may have been recalled without completing its objectives - in the case the crisis was less severe than expected.

As any KAOS model element, Objects may have model annotations, for example to indicate issues as in Figure 3.33. In addition, model annotations hold attributes of the object otherwise not shown in the diagram, as invariants of initialization values (Figure 3.34). For example, CrisisDetails should be initialized with the `Status` attribute set to `Active`.



Name CrisisDetails
Type Entity
Def
Has Identifier % Uniquely identifies the crisis. %
Time % Time at which the crisis has been reported. %
Status % States whether the crisis is resolved or not. %
Description % A textual description of the crisis. %
DomInvar Identifier must be defined.
Init Status = Active
Issue Special requirements regarding the identifier were not stated in the requirements document. If the system were to support handling of multiple crisis, there will be a need for an identifier usable by the coordinators, or an additional attribute.

Figure 3.34.: Features of `CrisisDetails` in model annotation

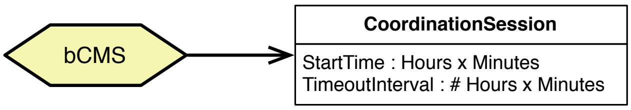

bCMS
CoordinationSession
StartTime : Hours x Minutes
TimeoutInterval : # Hours x Minutes

Figure 3.35.: `bCMS` controls `CoordinationSession`

In the next parts of the model the capabilities of the modeled agents are discussed with respect to the object model from above. `CoordinationSession` is entirely controlled by `bCMS` (Figure 3.35). `PSC` and `FSC` entirely monitor `TimeoutLog`, but only control the respective attribute holding the timeout reason (Figure 3.36). `bCMS` controls `Time` and `Date` of `TimeoutLog` and monitors both reason attributes.

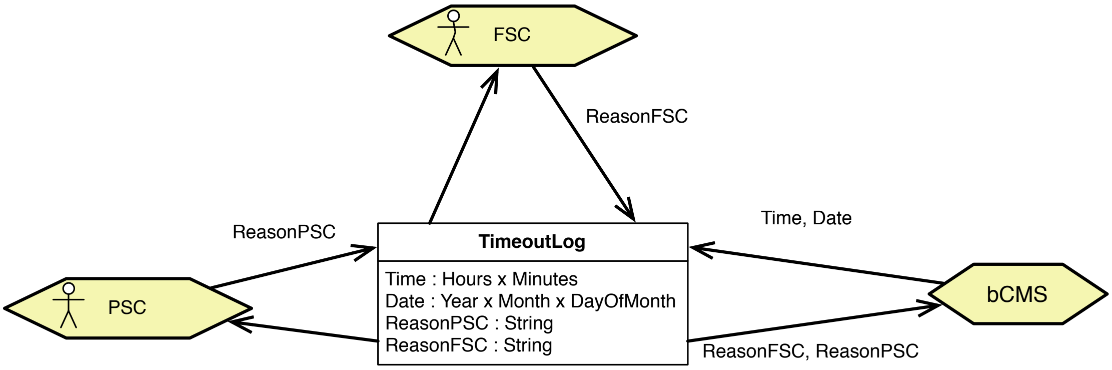


Figure 3.36.: Monitoring and Control of `TimeoutLog`

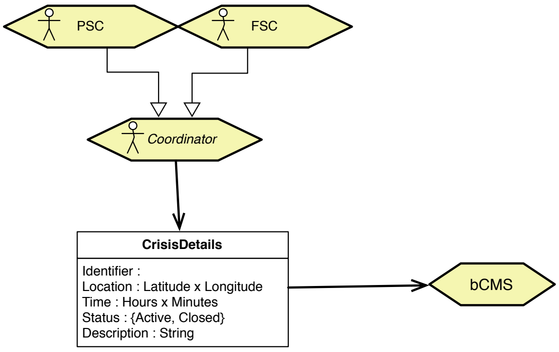


Figure 3.37.: Monitoring and Control of CrisisDetails

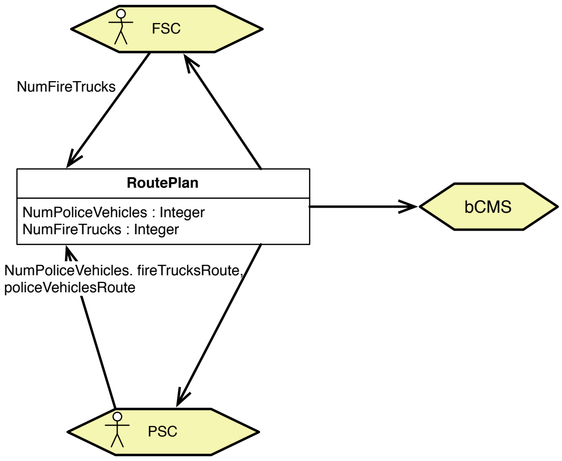


Figure 3.38.: Monitoring and Control of RoutePlan

Regarding the `CrisisDetails` entity there is no difference in capability between the two coordinators (Figure 3.37). Apart from that, it is not allowed in KAOS that two agents control the same attribute, association, or whole entity. Therefore, `Coordinator` is introduced as a superclass of `PSC` and `FSC`, which controls the `CrisisDetails`. `bCMS` only monitors the object. It is assumed that both coordinators collaboratively edit the `CrisisDetails`. This makes clear how the exchange of crisis details should be handled according to the current model.

`RoutePlan` is monitored entirely by all three agents. `NumFireTrucks` is controlled by the `FSC` and the other features by the `PSC` (Figure 3.38). In that current state of the model it is unclear how approval or disapproval of a fire truck route by the `FSC` will actually take place. This is related to the uncertainty about the nature of the communication channel between the two coordinators.

Of the `Route` entity, the `RouteState` attribute is controlled by one of the two coordinators (Figure 3.39). To indicate which concrete coordinator has control over which instance, an additional model annotation would be needed. An example for such an annotation is shown in Figure

3.40). The Path attribute is only controlled by the PSC, as the FSC may not define vehicle routes. bCMS is monitoring the whole entity.

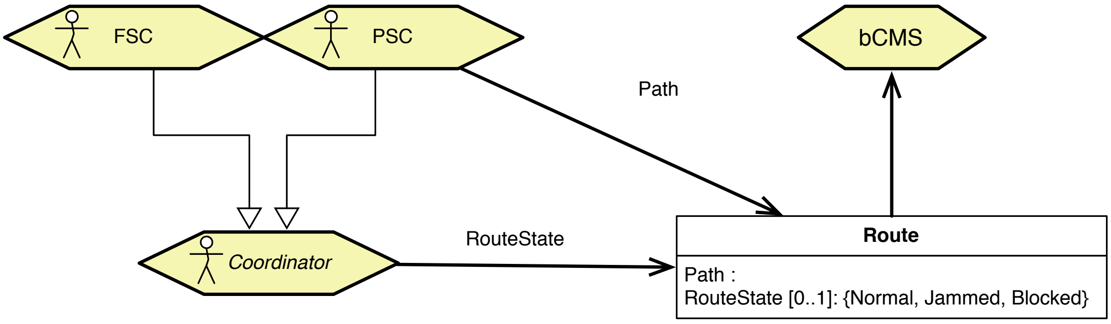
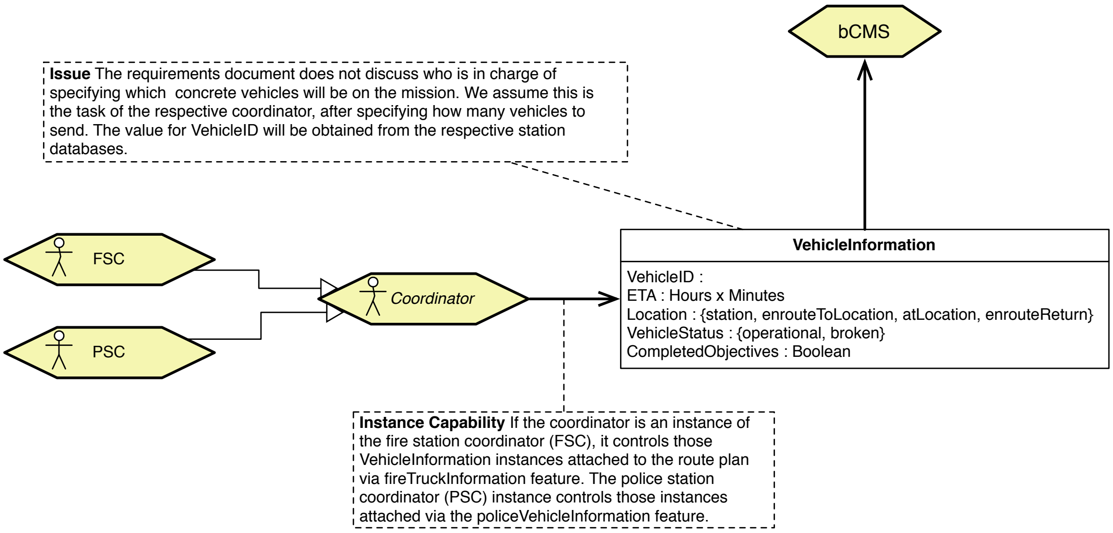


Figure 3.39.: Monitoring and Control of Route

The VehicleInformation entity is entirely controlled by a coordinator. In Figure 3.40 below an instance capability model annotation is used to explain that PSC and FSC only control those vehicle information instances that represent the information about those vehicles that the respective coordinator gets reports from .


Issue The requirements document does not discuss who is in charge of specifying which concrete vehicles will be on the mission. We assume this is the task of the respective coordinator, after specifying how many vehicles to send. The value for VehicleID will be obtained from the respective station databases.

Instance Capability If the coordinator is an instance of the fire station coordinator (FSC), it controls those VehicleInformation instances attached to the route plan via fireTruckInformation feature. The police station coordinator (PSC) instance controls those instances attached via the policeVehicleInformation feature.

bCMS
FSC
PSC
Coordinator
VehicleInformation
VehicleID :
ETA : Hours x Minutes
Location : {station, enrouteToLocation, atLocation, enrouteReturn}
VehicleStatus : {operational, broken}
CompletedObjectives : Boolean

Figure 3.40.: Monitoring and Control of VehicleInformation

With the agent capabilities defined, the operations with which the agents realize requirements or expectations can be modeled. As a first example, Figure 3.41 shows the CreateSession operation. The figure is also an example of how the modeling of one particular instance of a KAOS abstraction (an operation in this case) leads to the identification of another (an entity in this case): In order to describe the effects of the CreateSession, its inputs and outputs, and its trigger-, pre-, and post-conditions, we need to introduce a new entity, CoordinatorInfo, to the model. The object model for the new entity is shown in Figure 3.42.

`CreateSession` takes the `bCMS` and two `CoordinatorInfo` entities of `Connection` as input variables and creates a `CoordinatorSession` instance as an output variable, when the trigger condition is true. The trigger condition (`ReqTrig`) is shown in an annotation of the operationalization link between the requirement `Session Created` and the operation `CreateSession`. It states that the `Connected` association should hold between each of the two coordinators and `bCMS`.

The effect of the operation is described via its `DomPre` and `DomPost` attributes, stating that after performance of the operation, the `HasSession` association is present between the `bCMS` and the `CoordinationSession` instance. The operation performer of `CreateSession` is `bCMS`.

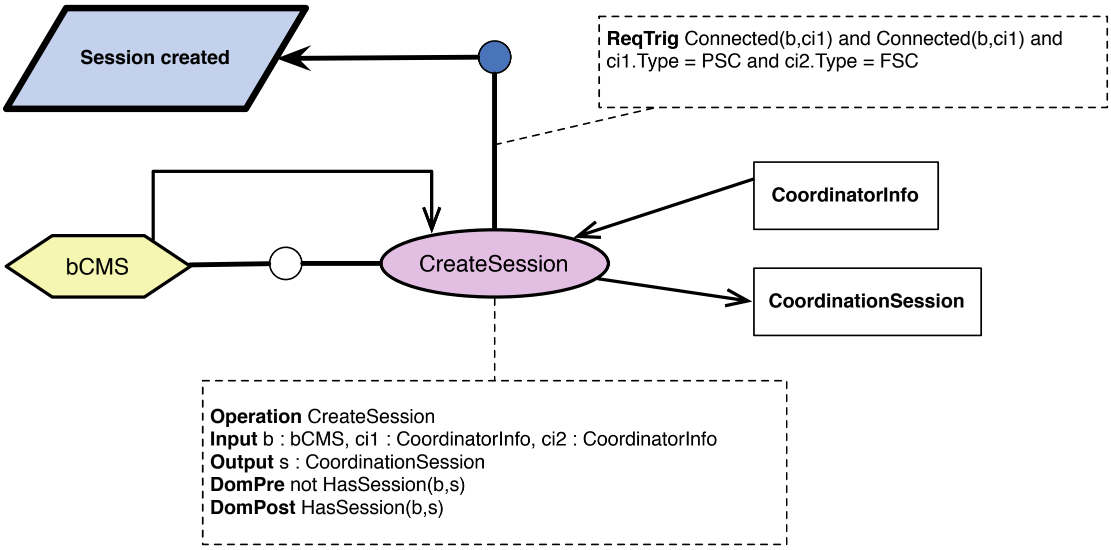

Session created
ReqTrig Connected(b,ci1) and Connected(b,ci1) and ci1.Type = PSC and ci2.Type = FSC
CoordinatorInfo
bCMS
CreateSession
CoordinationSession
Operation CreateSession
Input b : bCMS, ci1 : CoordinatorInfo, ci2 : CoordinatorInfo
Output s : CoordinationSession
DomPre not HasSession(b,s)
DomPost HasSession(b,s)

Figure 3.41.: `CreateSession` operation with performance, operationalization, and input/output links

An interesting observation is that the `CoordinatorInfo` entity appears in different notation on the two diagrams. When discussing operation input and output, a reduced notation is used, in which the entity's attributes are not shown within compartments of the entity rectangle but as labels of the input and output links. It is not clear whether both variants of representation may appear on the same diagram. As in KAOS agents are objects and associations relate objects, it is possible to integrate the software agent `bCMS` and the human environment agent `Coordinator` into the object model. `CoordinatorInfo` represents the knowledge the system has about the connected coordinator.

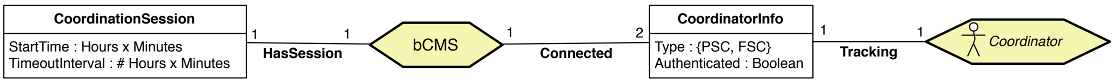


Figure 3.42.: Additions to object model to allow for the specification of `CreateSession`

The object model excerpt from above also supports the specification of the attributes of the ConnectPSC operation. It creates a software object CoordinatorInfo via which the bCMS is tracking the coordinator. When the CoordinatorInfo instance has been created, it is also related to the bCMS instance via the Connected association. Therefore, the ConnectPSC operation is a prerequisite for the CreateSession operation.

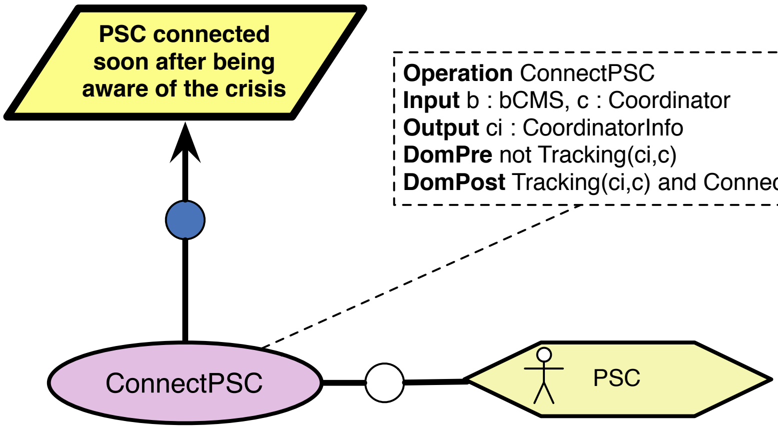

PSC connected soon after being aware of the crisis
Operation ConnectPSC
Input b : bCMS, c : Coordinator
Output ci : CoordinatorInfo
DomPre not Tracking(ci,c)
DomPost Tracking(ci,c) and Connected(b,ci) and ci.Type = PSC
ConnectPSC
PSC

Figure 3.43.: ConnectPSC operation

Figure 3.44 shows how the Password-based refinement of the Coordinators identified goal is obstructed by obstacles (compare Figure 3.25). The identification of coordinators cannot be achieved if at least one of the coordinators has forgotten the correct credentials needed for authentication. A higher-level obstacle is linked to Coordinators identified. If passwords were stolen by some malicious party, the identification of coordinators would not be achieved in a sense that the system cannot guarantee that actual coordinators are connected and identified. This diagram would need some elaboration in a later stage of the elicitation process, as Coordinators identified has multiple alternative refinements (as indicated in Figure 3.27). Only in systems in which the Password-based refinement is realized that obstacle has to be dealt with.

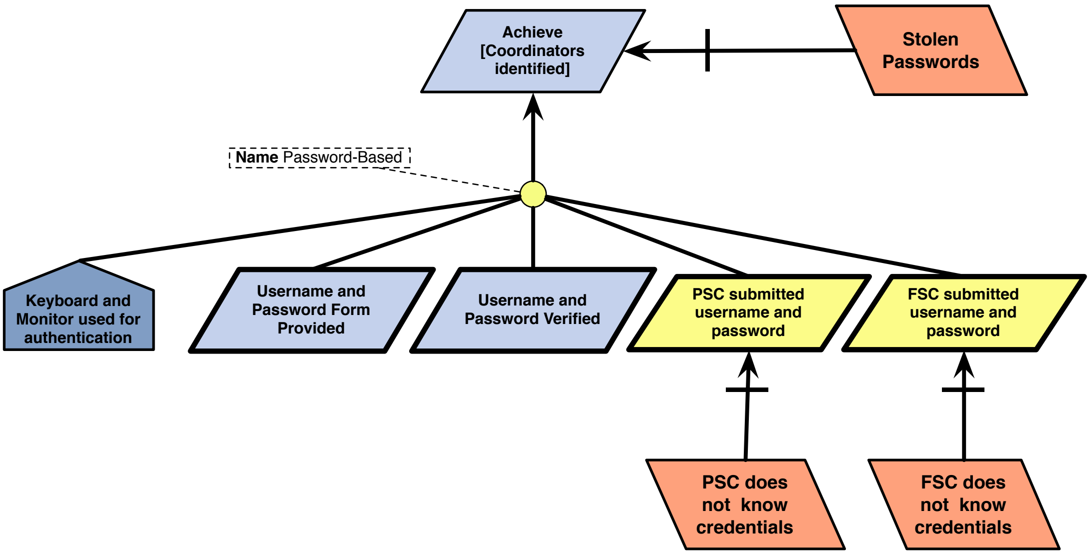

Achieve [Coordinators identified]
Stolen Passwords
Name Password-Based
Keyboard and Monitor used for authentication
Username and Password Form Provided
Username and Password Verified
PSC submitted username and password
FSC submitted username and password
PSC does not know credentials
FSC does not know credentials

Figure 3.44.: Obstacles to `Coordinators identified`

Resolutions to obstacles are again modeled as goals. In Figure 3.45, an accuracy goal called `Accurate Information about Fire Truck Dispatch` is refined into subgoals. One expectation towards the fire station coordinator is that he will avoid that any dispatch of a fire truck is not reported to the system. The accompanying expectation is that any dispatch will be reported as soon as the coordinator knows about it. Requirements towards `bCMS` are that the fire truck dispatch will be encoded in the system when the coordinator reports it and will be communicated over the communication channel, so that the other coordinator will eventually see the updated vehicle information. As the diagram is already complex, the agents were left out: the responsibility links from agents to leaf goals would interfere with the obstruction links between the obstacles and the leaf goals. Three obstacles are shown in the diagram, each obstructing a different leaf goal. One expectation towards the coordinator may not be fulfilled as there might be conditions under which the coordinator is unable to report, for example because of being distracted by some other events at the station. This is modeled in the obstacle `Dispatch not reported`. To account for the various reasons the coordinator might fail to report, the diagram can be elaborated so that `Dispatch not reported` is again refined into sub-obstacles, each one indicating one possibility. In this simpler variant of the diagram, distraction is assumed as the only option. The system might provide a resolution for this case by alerting the coordinator after a predefined interval to ask for the report.
The other two obstacles target possible failure modes of the system. `bCMS` might fail to record the dispatch report of the coordinator, for example because of a crashed hard disk. There is nothing the software can do about that, but the model can offer a resolution involving the environment of the system: it includes an expectation expressing that the system will run on resilient hardware, for example on a RAID system, in which failure of a certain amount of hard disks can be absorbed. The other obstacle is about

a failure of the communication channel, hindering the transmission of the information to the police station coordinator. In this case, the resolution is an expectation on the coordinators. As it is assumed that coordinators communicate with the field personnel (fire- or policemen) via other systems, it can be expected that the dispatch of a fire truck would eventually be communicated to the police station coordinator indirectly.

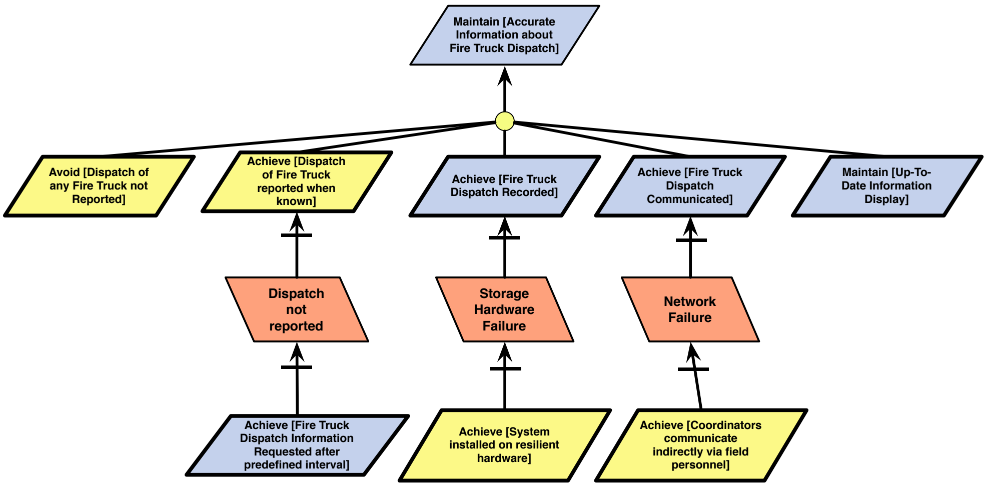

Maintain [Accurate Information about Fire Truck Dispatch]
Avoid [Dispatch of any Fire Truck not Reported]
Achieve [Dispatch of Fire Truck reported when known]
Achieve [Fire Truck Dispatch Recorded]
Achieve [Fire Truck Dispatch Communicated]
Maintain [Up-To-Date Information Display]
Dispatch not reported
Storage Hardware Failure
Network Failure
Achieve [Fire Truck Dispatch Information Requested after predefined interval]
Achieve [System installed on resilient hardware]
Achieve [Coordinators communicate indirectly via field personnel]

Figure 3.45.: Obstacles and resolutions

Apart from the fact that the presented model cannot be complete, as the requirements document is not complete as well, the model presented here does not represent all the information given in the requirements document. Furthermore, the model does not show all aspects of KAOS yet. Examples for various relationships are missing, like goal divergence (conflicting goals), goal concerns (goals' relations to objects), alternative agent assignments (between agent and goal), obstacle refinements (between obstacles), agent dependencies (involving multiple agents and goals). Examples of `SoftGoals` and how they guide the discussion of alternative goal refinements are not presented. No diagram touches upon the behavior modeling concepts of KAOS as state machines and scenarios in the form of message sequence charts. These two kinds of diagrams are well known from published modeling standards (OMG 2015c; ITU 2011).

3.1.7. Discussion

KAOS is a graphical requirements modeling language with a strong focus on the intentional view. It is implemented by one commercially available tool called Objectiver (Respect-IT 2014). While the meta-model provides a variety of abstractions, supporting not only the intentional view, but also a structural, a responsibility, a functional, and a behavioral view, the whole model is clearly goal-oriented. All requirements knowledge recorded in a KAOS model is driven by goals: `Goals` concern `Objects`, `Obstacles` obstruct `Goals`, `Agents` are responsible for `Goals`, `Agents` depend on other `Agents` because of `Goals`, `Operations` operationalize `Goals`, `StateMachines` and `Scenarios` cover `Goals`.

Variability is only supported in terms of Goal And/Or Graphs, constructed with the Refinement relationship. There are no dedicated abstractions for feature and product line modeling, different system variants can optionally be provided via textual `sysRef` annotations. Alternative goal refinements and actor responsibilities can be delineated from each other via model annotations. In our analysis it is not enough to discuss product variability with stakeholders as there is no model view from which it is easy to see the variability of a product line. Stakeholders, when interacting with the system, can be modeled as `EnvironmentAgents`. For other kinds of stakeholders not interacting with the system there is no dedicated abstraction in the language. They can only be hinted at via the `source` attribute of `Goal`. There is no dedicated abstraction for non-functional requirements, instead, the modeler can optionally classify a `Requirement` instance being non-functional via the optional `category` attribute inherited from `Goal`. Risk analysis is supported by modeling `Goal` obstruction with `Obstacles`.

The language has no public account of its revision history. This is detrimental for a precise understanding of the language. The language's meta-model has evolved since its first publication (Dardenne, Fickas, and Lamsweerde 1991), but no rationale is published regarding its evolution. It is not clear how the different sources describing aspects of the language relate to each other. For instance, the KAOS Tutorial (Respect-IT 2007) uses a diagram style whose abstraction level is comparable to the level of UML object diagrams (also known as instance diagrams) for describing the meta-model. Van Lamsweerde's textbook (Lamsweerde 2009) uses UML class diagrams. The KAOS Tutorial states that there "are two types of domain properties: domain hypotheses [...] (and) domain invariants [...]", whereas van Lamsweerde describes a slightly different taxonomy with "domain descriptions that are hypotheses or domain properties". The notation for domain descriptions, a trapezoid shape, is called "'home' shape" by van Lamsweerde, whereas the tutorial uses a pentagon resembling an idealized house which can also be interpreted as a "'home' shape". Van Lamsweerde's book uses only white and grey background colors for all shapes, whereas the KAOS tutorial uses more colors for the backgrounds. Cailliau et al. (Cailliau et al. 2013a) also use colors, but for some symbols use different colors than the tutorial. The notation for `DomHyp` (domain hypothesis) instances has changed, the background color is now different from the one for `DomInvar` (domain property) instances. The `SoftGoal` notation is also changed to a different shape, it now resembles a cloud symbol. Regarding the meta-model, a figure explaining the KAOS notation suggests that new abstractions were introduced to KAOS: Anti-Goal, Threat, Attacker, and Security Requirement.
A published language specification being under version control would be clearly preferable to the currently fragmented knowledge regarding KAOS.

The representation of the KAOS meta-model in (Lamsweerde 2009) is often incomplete. It is stated that every entity meta-class of the meta-model shares the attributes `name`, `def`, and issue. But there seems to be no abstract root class from which all meta-classes would inherit these attributes. For some meta-classes, these attributes are explicitly shown and for some not. The instanceVariable attribute of the meta-relationships Responsibility, Monitoring, Control, Performance, Input, and Output, is not shown on any diagram, "to avoid cluttering the diagram". The State meta-class seems to have subclasses, but they are not shown "for the sake of clarity". Several associations do not have multiplicities on all ends, and it is not clear wether 1..1 is the default multiplicity. No attribute of any meta-class is typed. Some meta-classes are shown without any attributes. The meta-classes BoundaryCondition, Goal, Operation are presented with a category attribute, but, as no type is given, we can only assume that their types are either specific enumerations or, if the respective class is at the root of a taxonomy, could be derived attributes. If they are enumerations, these are also not presented. The same criticism is valid for the category attribute of Agent, which is not even presented in the meta-model. For some classes it is not clear whether they can be instantiated, as no indication is given whether they are abstract or not. Examples are Goal, BehavioralGoal, LeafGoal, DomDescript or Object: their discussion by van Lamsweerde suggests these should be abstract.

In general, the KAOS meta-model needs many additional constraints to ensure correct instantiation. For example, the decomposition relationships of the State meta-class would allow for cycles in the graph if there is no constraint to enforce that a child state may not include any of its parent states. While the meta-model is specified using UML class diagrams, it does not use OCL to express and visualize any constraints within the meta-model. Some constraints could be avoided by a refined meta-model. For example, the Divergence meta-relationship models three different kinds of conflict: between goals, between goals and obstacles, and between domain descriptions and obstacles. In the current state of the model, constraints are needed to enforce correct cardinalities. In a refined model, with one meta-relationship for each kind of conflict, these constraints would not be necessary.

On two occasions, the meta-model is ambiguous: Responsibility and Instance meta-relationship appear multiple times on the same diagram. For two other aspects the discussion of meta-model parts in the book can lead to misconceptions about the meta-model: first the meta-model states that Goal has an optional category attribute. Its values are taken from a goal taxonomy that is not part of the meta-model. That taxonomy partitions goals into functional and non-functional goals. In the same section of the book, a very similar attribute named type is discussed that is, while not explicit in the meta-model, derived from parts of the Goal taxonomy that are present in the meta-model: its value can be one of {Achieve, Maintain, SoftGoal}, but not Requirement or Expectation. Another example is the DomDescript taxonomy: throughout the book "domain properties and hypotheses" are discussed: it seems DomDescript actually stands for "domain description" and is a super-class of "domain hypothesis" and "domain property". However, its subclasses in the meta-model are called DomHyp and DomInvar. So DomInvar, being described as "domain properties (called invariants)" must be taken as the meta-class for "domain property".

Minor issues of the meta-model are: some names of meta-classes and attributes are abbreviated, which makes them hard to read in the first place; for example, DomDescript, DomInvar, DomHyp, O-Refinement, def, or formalSpec. Usually, association names are uppercase and role names lowercase, but for the ObstructedBy role of the Divergence relationship that convention is violated. The Divergence relationship representation seems to have a typographic error: it is represented as a relationship named Divergenc. The stateVar role is only presented for Attribute meta-class but not for the Association meta-class, though it is described for both, mentioned as stateVariable.

A general observation regarding KAOS' meta-model is that it is not MOF-compliant, neither at EMOF or CMOF compliance level. It is violating various constraints common to EMOF and CMOF: It is violating a constraint that states "the option is disallowed of suppressing navigation arrows such that bidirectional associations are indistinguishable from non-navigable associations" (OMG 2015b). The constraint is violated for various associations, e.g. the OR-Ass association between Goal and Refinement in Figure 3.2. The meta-model also violates the constraint "a TypedElement other than a LiteralSpecification or an OpaqueExpression must have a Type". In the presentation of the KAOS meta-model in (Lamsweerde 2009), no meta-class attribute is typed. As class attributes are named Property in UML, and Property is a TypedElement, the constraint is violated. Both levels of compliance do not support the use of AssociationClass in meta-models. KAOS is non-compliant in that regard as it presents Link, Operationalization, and ValuesIn as an association class. Finally, the constraint "Property::aggregation must be either 'none' or 'composite"' is violated by the usage of aggregation in the meta-model, for example between Object and DomDescript.
Furthermore, the Divergence and Dependency meta-relationships violate an EMOF-specific constraint requiring that "an Association has exactly 2 memberEnds [...]". EMOF compliance is also violated through the use of "OR-Associations", which we interpreted as constraints. Constraints are only allowed for CMOF-compliant models.

We have found no concrete syntax meta-model for KAOS and also no rationale for the design of the notation. There are some differences between the notation presented in van Lamsweerde's book (Lamsweerde 2009) and the KAOS tutorial (Respect-IT 2007). The most important difference is the use of color in the tutorial. In the tutorial notational symbols can have much more different background colors, in the book there is only a difference between white and grey background. In the following discussion of the KAOS notation, we refer to both variants of a symbol, whenever the two notations differ and it is appropriate for the argument. In general, we assume the notation has not been designed with Moody's principles in mind.
KAOS has some issues regarding semiotic clarity. There are two cases of symbol redundancy: Agents have different symbols on scenario charts than on responsibility diagrams. And Entities are represented with a different symbol on agent diagrams than on object diagrams. In the grayscale variant of the notation the symbols for BehavioralGoals and LeafGoals introduce symbol overload, as the same symbol is used for Maintain and Achieve goals, and, respectively, for Requirement and Expectation. The box with dotted line border to represent model annotations is a case of symbol excess as there is no abstraction to correspond to the symbol. That there is no notation to represent BoundaryCondition instances that are not Obstacles is a case of symbol deficit.

To support the discussion of perceptual discriminability, we have analyzed the visual variables utilized by the KAOS notation in Table 3.1. KAOS symbols use shape, orientation, and texture, and in the colored variant also color. The single use of a discriminating sub-symbol, the stick figure enclosed in an `EnvironmentAgent` shape, is ignored in the table, as no other symbol makes use of such a sub-symbol. The color column refers to the values of the tutorial variant, in the grayscale version of the book the differences in color are much rarer. As the KAOS tutorial itself provides multiple possible background colors for domain property, we stick to the background color presented on the title page of the tutorial. The possible values for shape are Pentagon, Trapezoid, Rectangle, Hexagon, Parallelogram, Circle, Oval, and Rectangle with rounded corners. The orientation is either horizontal or vertical, for some symbols a direction can be inferred in addition. Background colors that are used are dark blue, white, yellow, black, light blue, red, and light purple. Texture is only used for the line borders of any symbol: a border can have a thin or a thick line which can be solid or dashed.

Any two symbols can differ in four visual variables at most. Leaving the maximum number of perceptible steps undefined, we can compute a naive form of visual distance measure by just counting the number of variables in which symbols differ. Following this simple metric, we can see that the `Entity` symbol is too close to the model annotation symbol and the `StateMachine` symbol, as it differs in only one visual variable. The primacy of shape is followed to some extent, as shape is the primary differentiator in the KAOS notation. However, many symbol shapes are quadrilaterals, so the difference in shape is often not very prominent. For `Obstacles`, even the same shape is used as for `Goals`, even though the abstractions are contrary in meaning. Redundant coding is used in the tutorial variant, where the background color supports the differentiation of symbols in addition to the shapes. Sometimes the color choice seems to follow no system, e.g. the `Operation` for which an `Agent` is responsible has a light purple background, but both `Agent` symbols have yellow backgrounds. The `Event` and `Expectation` symbols also have yellow backgrounds, so it could be inferred that yellow represents externality, would not the `SoftwareToBeAgent` also have that background color. As `Goal` symbols apart from `Expectation` have a blue background color, that might have been a better choice for `SoftwareToBeAgent` and `Operation`. Perceptual Popout is rare in KAOS. We have observed only four symbols with unique values: the background color of the `Obstacle` and `Operation` symbols, the stick-figure sub-symbol of `EnvironmentAgent`, and the shapes of domain property/`DomDescript` and `Operation`.

Regarding semantic transparency, most of the symbols in KAOS are semantically rather opaque than transparent. There are only very little cues about what a symbol could actually represent. A notable exception is the stick figure for `EnvironmentAgent`, but it is at the same time misleading as the `SoftwareToBeAgent` is represented by the same shape but without any cue. The red background color of the `Obstacle` shape (in the tutorial variant) is helpful for all readers with a cultural background in which red has a stop symbol connotation.

KAOS has several mechanisms for complexity management, e.g. the `Refinement` relationship for modeling `Goal` trees. Regarding cognitive integration, KAOS supports

Abstraction Name	**Shape**	**Orientation**	**Color (all with black border)**	**Texture**
Domain Property	Pentagon (T)Trapezoid (B)	vertical, bottom-up	dark blue background	solid thin line, solid background
Entity	Rectangle	horizontal	white / no background	solid thin line, solid background
EnvironmentAgent	Hexagon	horizontal	yellow background	solid thin line, solid background
Event	Pentagon	horizontal, left-to-right	yellow background	solid thin line, solid background
Expectation	Parallelogram	horizontal, left-to-right	yellow background	solid thick line, solid background
Final State	Circle	-	black background	solid thin line with offset, solid background
Goal	Parallelogram	horizontal, left-to-right	light blue background	solid thin line, solid background
Initial State	Circle	-	black background	solid thin line, solid background
Model Annotation	Rectangle	horizontal	white / no background	dashed thin line, solid background
Object	Rectangle	horizontal	white / no background	solid thin line, solid background
Obstacle	Parallelogram	horizontal, right-to-left	red background	solid thin line, solid background
Operation	Oval	horizontal	light purple background	solid thin line, solid background
Requirement	Parallelogram	horizontal, left-to-right	light blue background	solid thick line, solid background
SoftGoal	Parallelogram	horizontal, left-to-right	white background	dashed thin line, solid background
SoftwareToBe Agent	Hexagon	horizontal	yellow background	solid thin line, solid background
State	Rectangle with rounded corners	horizontal	grey background	solid thin line, solid background
State Machine	Rectangle with rounded corners	horizontal	white / no background	solid thin line, solid background
Timeline	Rectangle	vertical, top-down	grey background	solid thin line, solid background

Table 3.1.: Visual Variables used by KAOS Notation

conceptual integration with context diagrams. But the notation has very little support for perceptual integration mechanisms. If, for example, a goal tree is large and thus split up into multiple diagrams, KAOS provides no visual cues to navigate between the split diagrams. There is no notation to visualize the `Class-` and `InstanceCoverage` relationships. To the best of our knowledge, only the `Episode` notation for scenario charts represents a visual cue, to highlight that a specific part of the scenario is only visible in another diagram.

The visual expressiveness of KAOS could be improved by using more information-carrying visual variables. Currently, shape is predominantly used to distinguish between abstractions. For some symbols orientation or texture support the distinction. Color is only used redundantly, it only augments the perceptible difference already established by the other three variables. So KAOS is not visually one-dimensional, but also far away from being visually saturated. This could be improved by establishing a consistent approach to using color in the notation, with the colors carrying meaning. Currently, the use of color only for the background color of the shapes also reduces drawability of the notation. Instead, the borders could be colored.

Dual Coding is rarely used in the KAOS notation: Only for `BehavioralGoal` instances text is used to differentiate between `Maintain` and `Achieve Goals`. Regarding graphic economy, our analysis is that the notation has a comparably low graphic complexity, for example compared to UML: Moody states that UML class diagrams have a graphical complexity of 40 (Moody 2009), and UML has much more diagram types. For KAOS we have counted 18 distinct graphical symbols for meta-entities and 11 distinct graphical symbols for meta-relationships[7]. The optional visualization of attributes within model annotations supports lowering the graphic complexity. It could be improved by eliminating symbol redundancy and more visual expressiveness.

KAOS does not support cognitive fit, i.e. there are no dedicated dialects of the notation for different purposes, e.g. for novices or for drawability.

3.2. User Requirements Notation (URN)

URN is a graphical requirements modeling language published by the Telecommunication Standardization Sector of International Telecommunications Union (ITU-T). It is targeted towards describing the requirements and the high-level design of "most types of reactive systems and information systems" (ITU 2012, pp. v, 1). URN was first published as an ITU standard in 2008 (ITU 2008a). At present, its second major revision is in force (ITU 2012), which is the version discussed in this section. URN has been influenced by three different research approaches(ITU 2008a, pp. v, 1): i* (Yu 1997), the NFR framework (Chung et al. 2000), and use case maps (Buhr and Casselman 1995). Information on the history of URN can be found in (Amyot and Mussbacher 2011). To the best of our knowledge, URN is currently implemented in one open source tool, jUCMNav (uOttawa 2001).

[7] In this dissertation the focus is on the notation for the meta-entities. A preliminary analysis of the KAOS relationship notation is presented in Appendix Section A.9

URN consists of two sub-languages, the Goal-Oriented Requirements Language (GRL) and Use Case Maps (UCM). GRL focuses on modeling stakeholders, their intentions, and the relationships between stakeholders and intentions. UCM focuses on the modeling of system behavior on a high level, which is modeled in terms of paths, components, and responsibilities. The two sub-languages share a common language core (Fig. 3.46). The language core partitions a URN specification, represented by URNspec[8], into two sub-specifications, a UCM specification, represented by UCMspec, and a GRL specification, represented by GRLspec. The separation of the two sub-languages is also reflected in the URNmodelElement taxonomy: UCM specifications contain only UCMmodelElements, and GRL specifications contain only GRLmodelElements. A URN specification contains URNlinks which are binary relationships relating two URNmodelElements. URNlinks have semantics beyond linking URNmodelElements of any kind but have a type attribute to carry a user-defined meaning. As opposed to URNmodelElement, URNlink is not the root of an inheritance hierarchy, i.e. it is not subclassed by UCM- or GRL-specific relationship meta-classes. In addition to the URNlink, there can be a loose coupling of elements from both sub-languages via Concerns. A Concern provides a "unit of understanding", potentially grouping elements from both UCM and GRL specifications.

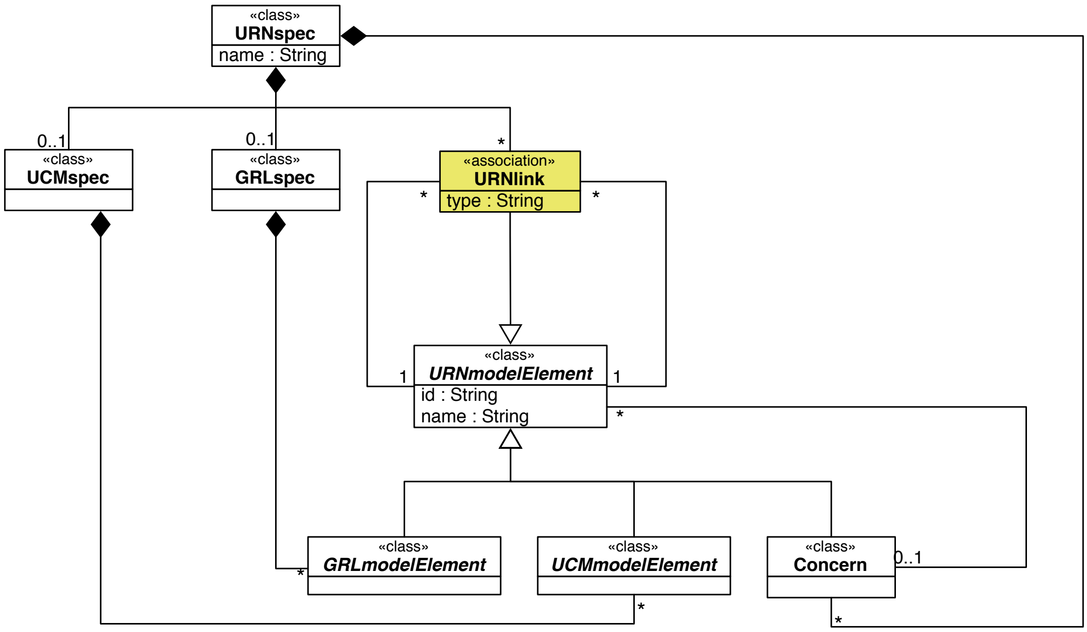


Figure 3.46.: The top-level abstractions of URN. Adapted from (ITU 2012).

In the following subsections the GRL (Section 3.2.1) and UCM (Section 3.2.2) are

[8] The URN specification follows a special camel-case convention for meta-class names. If a name starts with an all-uppercase acronym, the first letter following the acronym is a lowercase letter, e.g. as in URNspec or URNmodelElement.

described in more detail. This is followed by a critical discussion of the language in Subsection 3.2.4.

3.2.1. GRL

The main abstractions of GRL are `Actor` and `IntentionalElement`. `Actor` represents an entity with intentions. It may be a human, an organization, the system under discussion itself or one of its parts, or any other kind of system interacting with the system under discussion.
The intentions of an actor are expressed via different kinds of `IntentionalElements` (ITU 2012, pp. 23-24): a `Goal` is a "condition or state of affairs in the world" that an actor wants to achieve. There should be clear-cut criteria for determining whether the goal is achieved. Opposed to the `Goal`, a `Softgoal` is also a condition but without "clear-cut criteria for whether the condition is achieved". A `Task` is "a particular way of doing something". It usually is the operationalization of a `Goal` or a `Softgoal`, but it can be modeled without the operationalized goal being known. A `Resource` is some entity "for which the main concern is whether it is available". A `Belief` provides background knowledge for other parts of the model.
`Indicators` allow to incorporate real-world measurements into the model. `Indicators` are contained within `Actor` definitions, because they describe "actor-specific measurements" (p. 38).
`Indicators` and `IntentionalElements` can be referred to by an `Actor`. This commonality is modeled by their superclass `GRLContainableElement`. All three main abstractions of GRL (see Fig. 3.47) share the superclass `GRLLinkableElement`. `GRLLinkableElement` provides features to make its instances linkable, i.e. being participants in an `ElementLink` association.

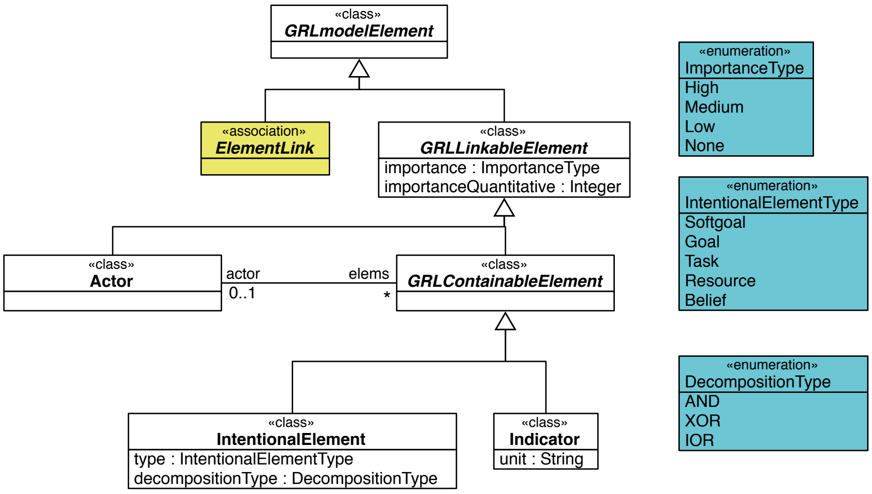


Figure 3.47.: The main entity concepts of GRL. `ElementLink` is included to provide context. Adapted from (ITU 2012).

GRLLinkableElements can be linked by ElementLinks of three different types: Decompositions, Contributions, and Dependencies (see Fig. 3.48). The main difference between these types is the kinds of entities they can link. This is specified by the specific constraints of each ElementLink subclass[9]. Decompositions are conditional relationships between GRLContainableElements and IntentionalElements. Together with the decompositionType attribute of a target IntentionalElement a set of decomposition links specifies what source elements need to be satisfied in order to satisfy the target. Decomposition is constrained so that Actors and IntentionalElements of type Belief may not be source or destination of it.
Contributions are links between GRLContainableElements and IntentionalElements. Contributions express how a GRLContainableElement affects the satisfaction of an IntentionalElement. A special subtype of Contribution, indicated with a boolean flag, is called correlation link and is only possible between intentional elements. It describes side-effects of intentions. The constraints of Contribution exclude Actor from being source or target of a Contribution. Furthermore, Belief, Resource, and Indicator may not be destinations of a Contribution. If the Contribution is a correlation, then Indicator may not be a source.
Dependencies model how an Actor can achieve its goals with the help of another Actor. For a more detailed model of a dependency between actors consecutive Dependency links are used. This way, it can be shown how a *depender* depends on a *dependee* for a *dependum*. The depender is an actor that is the source of the first link, or an actor containing the source IntentionalElement of the first link. The dependee is an actor that is the destination of the second link, or an actor containing the destination IntentionalElement of the second link. The dependum is an intentional element, which is not inside an actor definition and which is the destination of the the first link and the source of the second. Dependencies can also be modeled partially, as either depender, dependee, their intentions, or the dependum might be unknown. The constraints of Dependency exclude Belief from being part of a dependency. Furthermore, a Dependency may not link two elements that are both not contained in an Actor. It may also not link two elements of the same Actor.

[9] These constraints are not shown in the class diagram of Fig. 3.48, but they are explained in the following.

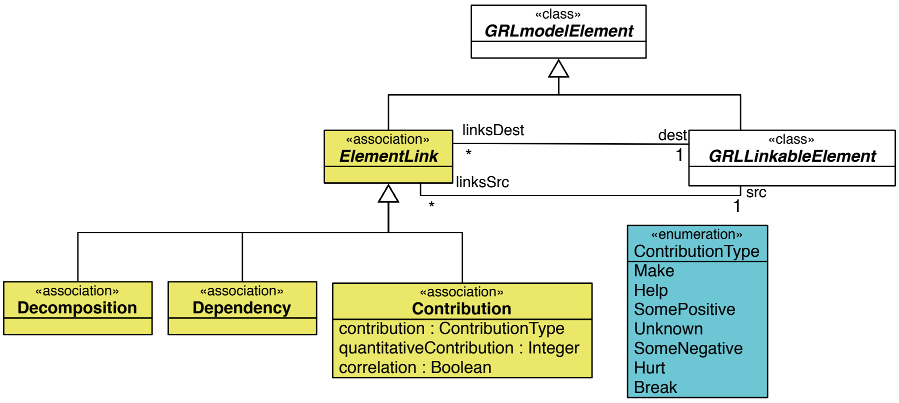


Figure 3.48.: The main relationship concepts of GRL. `GRLLinkableElement` is included to provide context. Adapted from (ITU 2012).

To support the evaluation of satisfaction levels of GRL models, two attributes of `GRLLinkableElement` are used: `importance` and `importanceQuantitative` (See Figure 3.47). The `importance` attribute can be used to define *qualitative importance*, using the ImportanceType enumeration, that has the literals None, Low, Medium, and High. Quantitative importance can be defined by the integer attribute `importanceQuantitative`, with values being constrained to the interval [0..100], indicating no importance (value 0) to high importance (value 100). Which of the two is used depends whether the importance analysis of the model is done qualitatively or quantitatively. If both kinds of analysis are desired, the two attributes should be kept consistent, but there is no constraint to enforce this. The semantics of the two attributes is defined in `Actor` and, respectively, in `GRLContainableElement`. The importance of an `Actor` is to be interpreted as importance "to the overall GRL model" and the importance of `GRLContainableElements` as importance to the containing `Actor` instance. To re-iterate, the two importance attributes express that different elements are of different importance to certain actors and different actors are of different importance to the GRL model.

Evaluation of GRL models is enabled by the `Evaluation` and `EvaluationStrategy` meta-classes (Fig. 3.49). The concrete evaluation algorithms to work on a GRL model cannot be described within GRL. The URN specification only provides some guidelines how such algorithms should work on a GRL model. Before startup of the algorithm, the model should contain at least one `EvaluationStrategy` and for a subset of `GRLContainableElements` corresponding `Evaluation` instances. Each of these `Evaluations` should have, depending on whether the evaluation algorithm uses quantitative, or qualitative values, or both, at least one of its two `evaluation*` attributes set. The `exceeds` attribute shall be set if the algorithm considers exceedance when evaluating goal satisfaction. These three attributes provide initial evaluation values, which will be propagated through the GRL model by the evaluation algorithm. The GRL specification is incomplete on that matter: it states that the initial values shall be copied to attributes of the related `GRLContainableElements`, but the specification defines no suitable attributes for that purpose. It is also unclear whether there shall be a 1:1 mapping of

GRLLinkableElements to Evaluations. We assume that this is not the case, as the specification is stating that GRLLinkableElements, which have been declared as not important by the modeler, may be excluded from evaluation. For any importance value higher than None (or 0, respectively) the exact impact of the importance value to the satisfaction level is up to the concrete evaluation algorithm, as well as the exact role of the ElementLink subclasses in the propagation of evaluation values. A special role in evaluation is assigned to Indicators. As the initial values assigned to the Evaluations are based on the subjective judgement of the modelers, the evaluation of the whole GRL model can be considered being subjective. The basic idea of Indicators is that for certain goals supporting evidence in the form of real-world measurements already exists. These real-world measurements can be recorded in the model in IndicatorEvaluation instances. The meta-model for IndicatorEvaluation is irritating at first sight, as IndicatorEvaluation is no subclass of Evaluation but optionally contained in an Evaluation. Furthermore, it has no direct relationship to the Indicator, which is puzzling, as there is also no constraint that would make a contained IndicatorConversion only admissible for Evaluations that have an Indicator as the evaluated element.

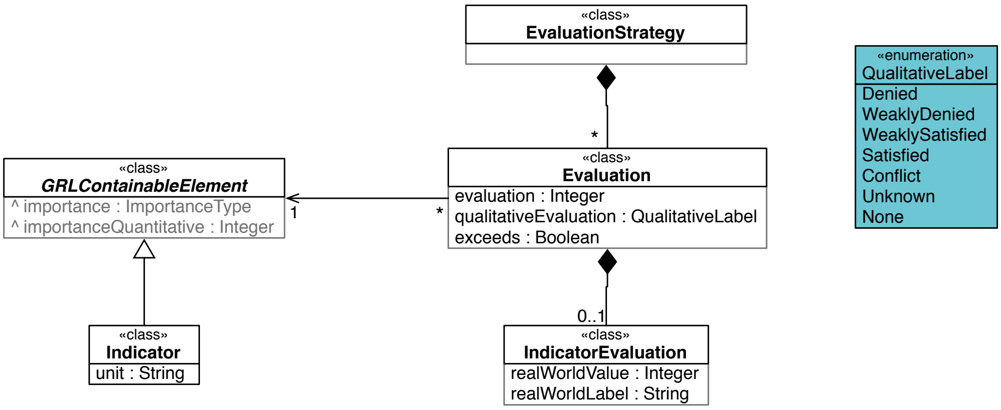


Figure 3.49.: GRL meta-model support for evaluation algorithms. Adapted from (ITU 2012).

3.2.2. UCM

The main abstractions of UCM are Responsibilities, Components, and UCMaps (see Fig.3.50). The central abstraction is the *use case map*, represented by the UCMap metaclass. A use case map is used to model a *scenario*. In the context of URN a scenario is defined as “a partial description of system usage defined as a set of partially-ordered responsibilities a system performs to transform inputs to outputs while satisfying preconditions and postconditions” (ITU 2003). We will review this definition of scenario and how it is matched by the UCM meta-model of URN, after we have introduced more of its abstractions.

Component is an abstraction to represent entities with behavior, which can be parts of the system under discussion, but also entities of the environment, including humans.

Responsibility is an abstraction modeling “a reusable definition of a scenario activity representing something to be performed“ (ITU 2012).

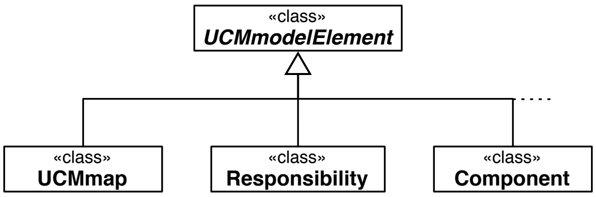


Figure 3.50.: The main concepts of GRL. The dotted line indicates that UCM contains more subclasses of UCMmodelElement. Adapted from (ITU 2012).

Another central notion of UCM is that of a *path* which has no direct counterpart meta-class in the meta-model. A path is expressed by several constituents of a use case map. A use case map can contain multiple of these paths describing partial behavior of the system under discussion, but also of the environment in which the system is embedded. The behavior expressed by a use case map can be understood by following the paths expressed in the map. A path consists of nodes (PathNodes) and connections between these nodes (NodeConnection). The NodeConnection connects a source PathNode with a target PathNode (see Fig.3.51). It thus establishes a directed graph. Components are not directly contained in use case maps, but are referred to by dedicated UCMap parts, the ComponentRefs. The ComponentRefs structure the use case map, as each ComponentRef on a use case map refers to a subset of the PathNodes contained in the map. Notationally this is represented in the following way: the PathNodes linked to a ComponentRef are placed within the bounds of the CompontRef's graphical representation. To allow for iterative refinement of a use case map, the PathNodes of a map are not necessarily bound to a ComponentRef: a use case map may only express some paths and have no information about components, before ComponentRefs are introduced to highlight the Components in which that part of the path (i.e. partial behavior) takes place. What behavior is exactly entailed by such an partial path is defined by the concrete subclasses of PathNode.

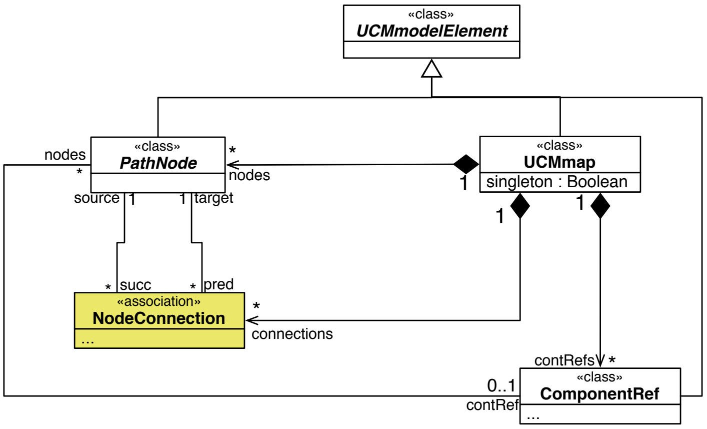


Figure 3.51.: The basic constituents of a map. Adapted from (ITU 2012).

The taxonomy of `PathNode` is shown in Figure 3.52, along with the links to `Responsibility` and `Component`. The concrete `PathNode` subclasses define the possible semantics of the path. Depending on what kind of `PathNode` is used, the path starts, ends, forks, joins, waits for a condition to be fulfilled or for time to pass, fails, connects to another path, refers to another map, or indicates a `Responsibility`. How the `PathNode` subclasses listed in Figure 3.52 are represented on a UCM diagram, can be seen at the end of this section, in Figure 3.53. It represents a use case map without `ComponentRefs` and `NodeConnections`.

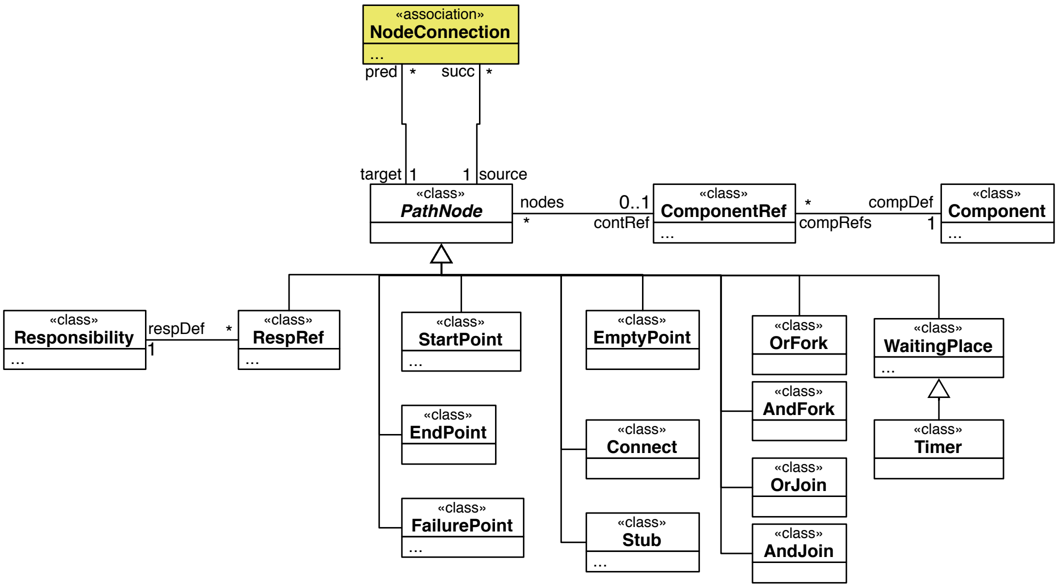


Figure 3.52.: Kinds of nodes. Note that there is no meta-class representing the whole path. Adapted from (ITU 2012).

A path is started by a StartPoint node and ended by an EndPoint node. It can fork in two different ways: AndFork nodes constitute the beginning of concurrent actions. OrFork nodes stand for the choice of a subset of paths starting from the fork node (i.e. only the actions on a subset of the paths occur). The forked subpaths can be joined again by either OrJoin or AndJoin nodes. OrJoins indicate that a subset of the joined subpaths has to fulfill a certain condition before the path may be continued. AndJoins indicate that all incoming paths have to be synchronized before the path can be continued. A WaitingPlace is a node in the path that indicates that the scenario may only be continued if a condition is fulfilled or an event has occurred. A Timer is a specialization of WaitingPlace that also allows scenario continuation after a timeout event. A FailurePoint node establishes a path branch which is followed if a failure condition holds. If the failure condition does not hold, the branch of the path specifying regular behavior is followed.

Connect, EmptyPoint, and Stub nodes have the purpose of structuring complex scenarios. Connect nodes allow for synchronous or asynchronous connections between paths. If the connection is asynchronous, an EmptyPoint is the source of the NodeConnection that has Connect as target node. If it is synchronous, an Endpoint is the source of such a NodeConnection instance. Using Connect, the modeler can describe local scenarios within a Component. Stubs allow for hierarchical structuring of use case maps. If it is used on a path, it refers to *plug-in maps* defined elsewhere. In the case of a static Stub, there is at most one plug-in map. Dynamic Stubs can have more than one plug-in map, which plug-in maps are executed depends on a condition to be specified by the modeler. A synchronizing Stub is a special kind of dynamic Stub, which requires all plug-in maps for which the entry condition was true to complete before the next node after the Stub can be reached. A blocking Stub is also a special kind of a dynamic Stub, which allows traversal of its plug-in maps exactly once. Using Stubs, the modeler can simplify paths by extracting parts of the path into a plug-in map. If that is necessary for some reason, the plug-in maps can also be re-integrated into the main map.

The RespRef subclass of PathNode refers to a Responsibility, specifying what the corresponding Component does at that point of the scenario. By ordering the PathNodes through the NodeConnection relationships, causal relationships between the Responsibilities are visualized. In addition, a map can contain references to Components. By superimposing paths and thus Responsibilities on ComponentRefs, a map not only visualizes what happens in a scenario, but also under which conditions a specific Component participates in the scenario at which point in time.

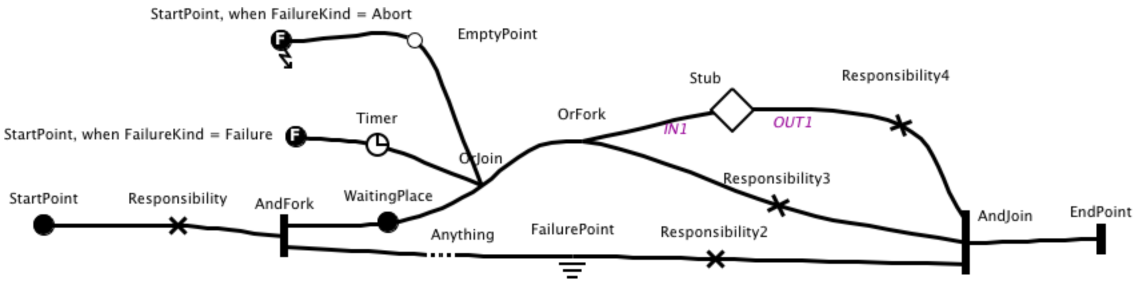


Figure 3.53.: `PathNode` and `NodeConnection` notation. Created with jUCMNav.

3.2.3. Example Model

As in the section on KAOS, the usage of URN is illustrated by means of the bCMS example problem. The diagrams shown in this section were created with the jUCMNav CASE tool. As it does not support the collapsed actor notation of URN, dependencies between actors have to be shown with diagrams where the expanded actor notation is used. Again, the example model is based on our own judgement, to gain greater flexibility regarding the choice which kinds of diagrams to include and which not. A comparable model of the bCMS problem expressed in i* has been done by Horkoff (Horkoff 2012).

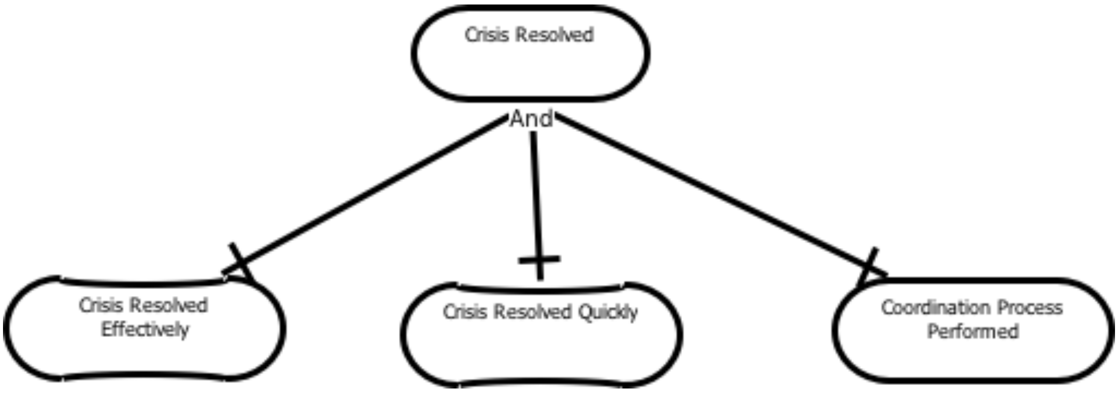


Figure 3.54.: Decomposition of `Crisis Resolved`

`Crisis Resolved` is decomposed into two soft goals, `Crisis Resolved Effectively` and `Crisis Resolved Quickly`, and one hard goal, `Coordination Process Performed` (Fig. 3.54). The two soft goals are modeled as such as there is currently no known hard criterion regarding effectiveness and speed of the crisis resolution. It is, however, possible to model how certain other goals may contribute positively or negatively to these soft goals. In order to really resolve the crisis, more goals need to be achieved, e.g. that firemen arrive as quickly as possibly at the location. As the scope of this example model is bCMS, these goals are omitted. The focus is completely on the coordination process between police station coordinator and fire station coordinator.

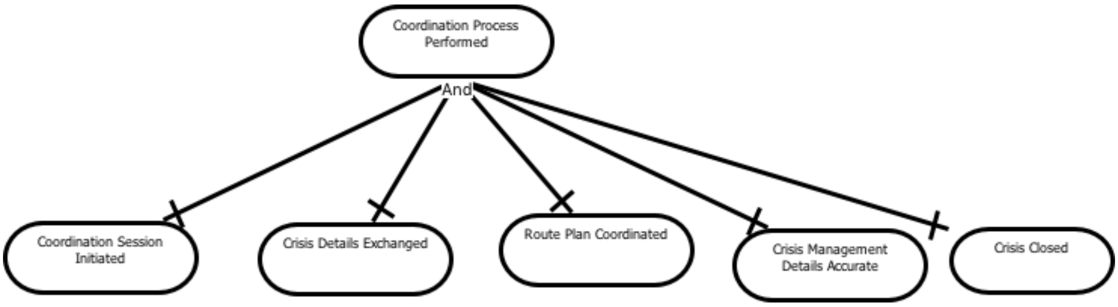

Coordination Process Performed
And
Coordination Session Initiated
Crisis Details Exchanged
Route Plan Coordinated
Crisis Management Details Accurate
Crisis Closed

Figure 3.55.: Decomposition of `Coordination Process Performed`

Similar to the KAOS model `Coordination Process Performed` is broken down into five subgoals, `Coordination Session Initiated`, `Crisis Details Exchanged`, `Route Plan Coordinated`, `Crisis Management Details Accurate`, and `Crisis Closed` (Fig. 3.55). `Coordination Session Initiated` is discussed in more detail in the following paragraphs. First, it has to be achieved that both coordinators are connected, so that they are able to identify themselves (Fig. 3.56) . Then, bCMS is responsible for verifying the identity of the two coordinators.

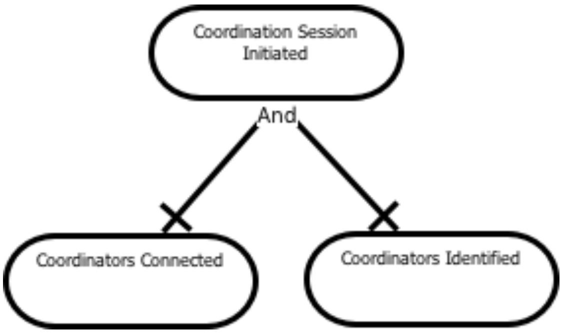


Figure 3.56.: Decomposition of `Coordination Session Initiated`

In Figure 3.57 the intentional elements that satisfy `Coordinators Connected` are allocated to the aforementioned actors. `Coordinators connected` is modelled as a goal of `bCMS`. Even though it may be a goal of the coordinators too, or some yet unmodeled stakeholder, the goal in the context of this example model is only to be achieved in the scenario in which `bCMS` exists. In a `pre-bCMS` world, where the coordinators are using another system, a comparable goal could have been "Coordinators in telephone call".

`Coordinators Connected` is decomposed into three tasks in this figure: both coordinators should connect to `bCMS`, and `bCMS` has to create a session in which the exchange of information between the coordinators is managed.

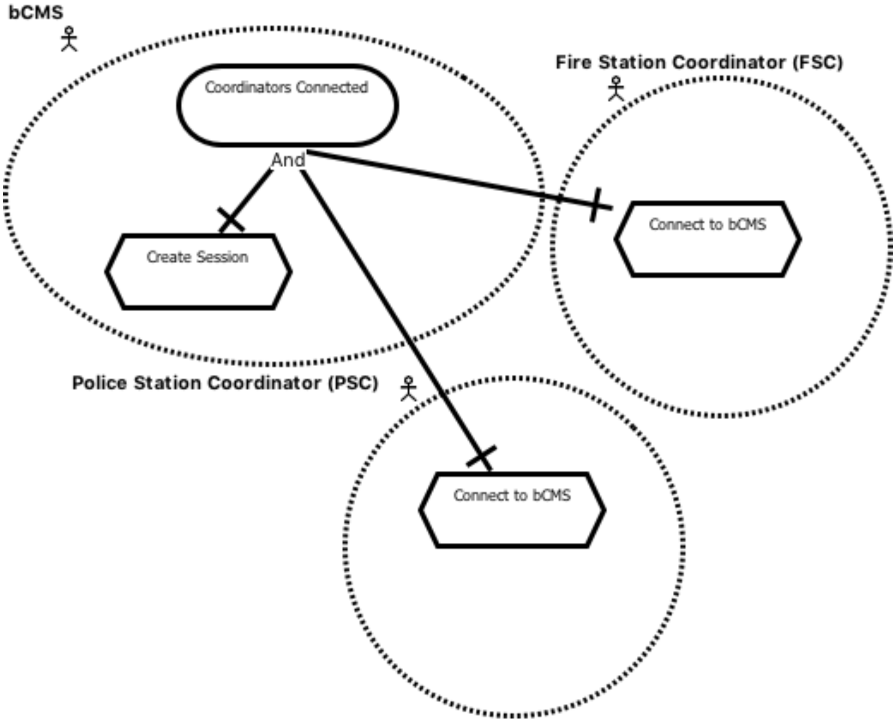


Figure 3.57.: Decomposition of `Coordinators Connected`

To identify the coordinators, an authentication of the coordinators is to be done by `bCMS` - in a reliable and safe manner, so that no unauthorized person can participate in the coordinaton process. In Figure 3.58, `Coordinators Authenticated` is still a rather abstract goal, even though it is decomposed into the tasks `Provide Credentials` and `Verify Credentials`. The two tasks are themselves rather abstract and need further refinement.

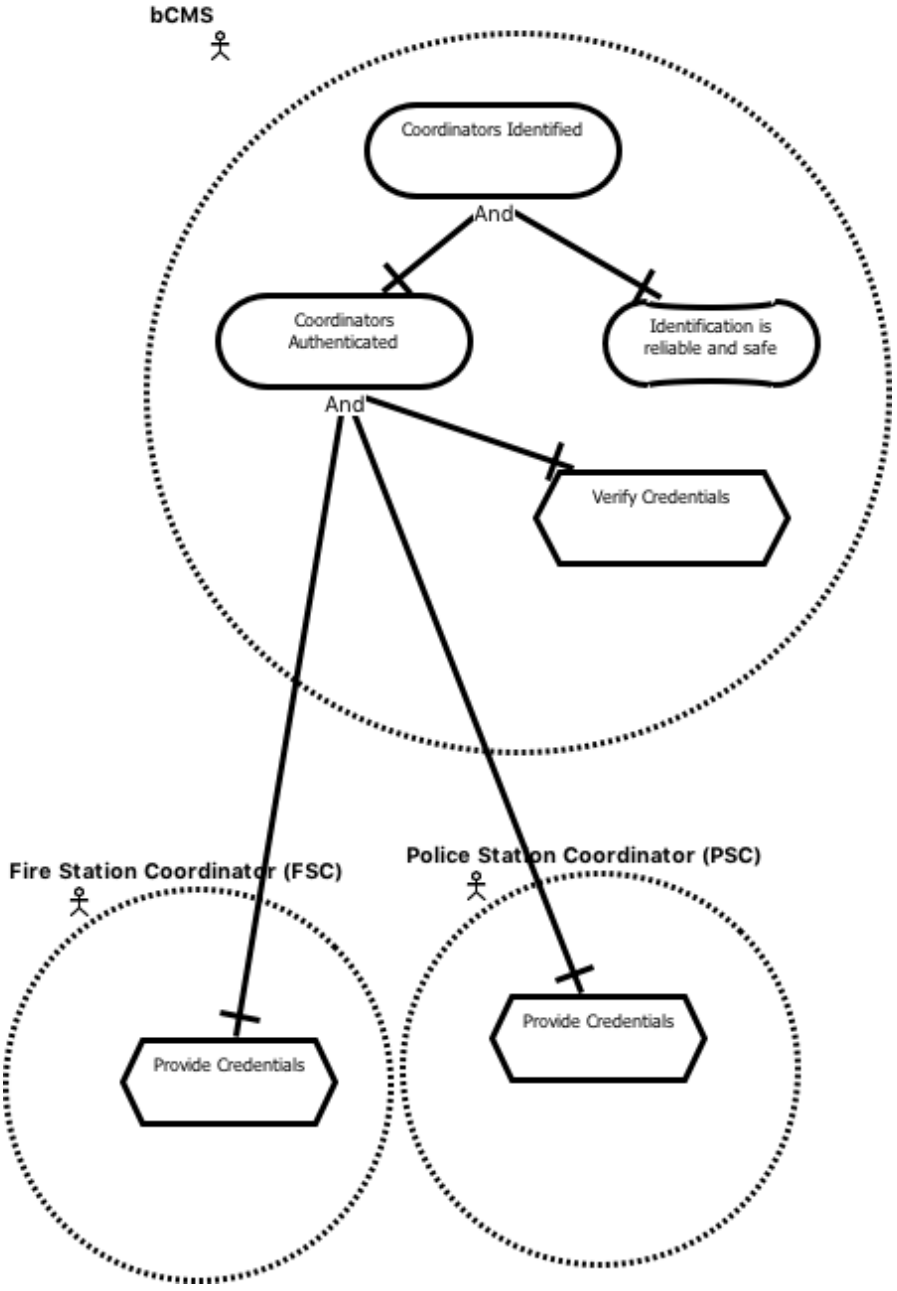

bCMS
Coordinators Identified
And
Coordinators Authenticated
Identification is reliable and safe
And
Verify Credentials
Fire Station Coordinator (FSC)
Police Station Coordinator (PSC)
Provide Credentials
Provide Credentials

Figure 3.58.: Decomposition of `Coordinators Identified`

Figure 3.59 presents some details about these two tasks by Or-Decomposing them. In terms of the meta-model, the `decompositionType` attribute of the two `IntentionalElement` instances `Provide Credentials` and `Verify Credentials` is set to `OR`. Both decompositions together outline three different but not mutually exclusive ways to achieve the authentication goal. The figure also presents a dependency expressing that `Ver-`

ify Credentials, and thus its containing actor bCMS, depends on the task Provide Credentials contained by the FSC and PSC actors.

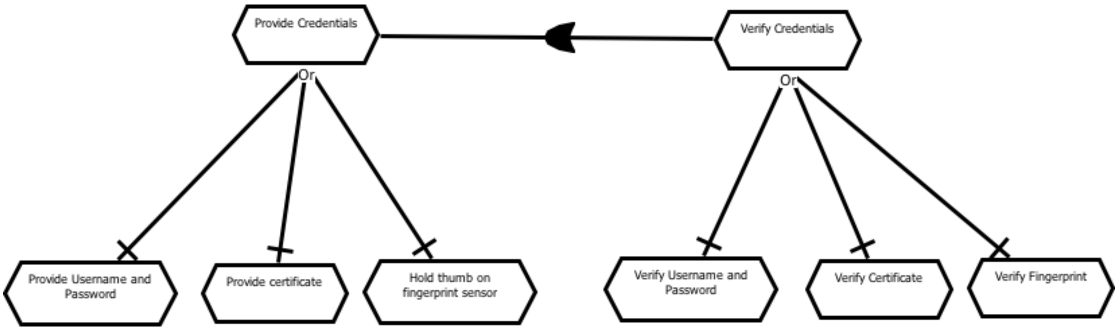


Figure 3.59.: Or-Decomposition Provide Credentials and Verify Credentials

A more detailed perspective on inter-actor dependencies is given in Figure 3.60. By showing the containment of the IntentionalElement instances in the respective actors, together with the dependency links between them, the figure defines how bCMS depends on the two coordinators regarding verification of credentials, and how both coordinators depend on a resource, in this case that a communication channel that needs to be available. This resource is modeled outside of the three actors as it belongs to another actor, the hardware platform that bCMS is running on.

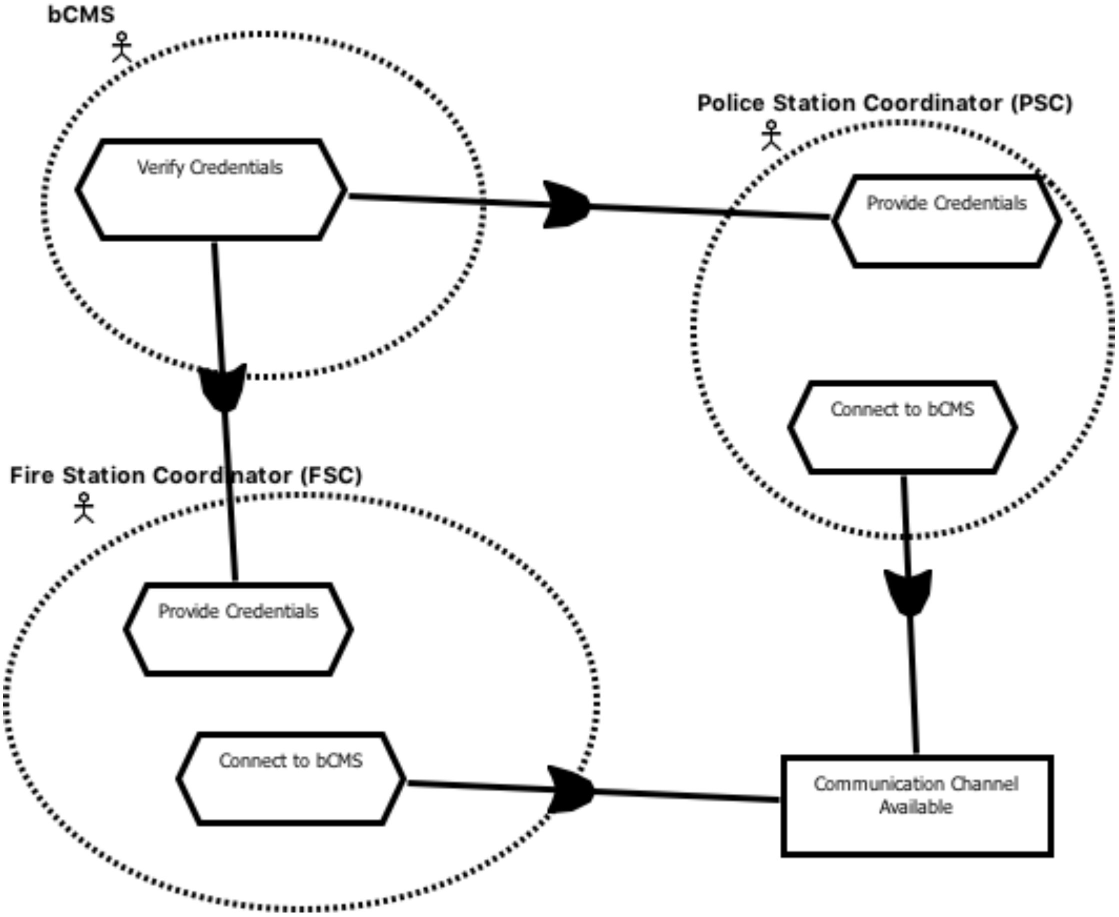

bCMS
Verify Credentials
Police Station Coordinator (PSC)
Provide Credentials
Connect to bCMS
Fire Station Coordinator (FSC)
Provide Credentials
Connect to bCMS
Communication Channel Available

Figure 3.60.: `Dependency` Relationships

After both coordinators are connected and authenticated, the next goal to achieve is `Crisis Details Exchanged` (Fig. 3.61). It is decomposed into two tasks for bCMS, `Merge Crisis Knowledge`, and `Present Crisis Knowledge`, and two tasks `Provide Crisis Knowledge`, one for each of the two coordinators. It is assumed here that `bCMS` will be able to deal with conflicting information given by the two coordinators when merging the information.

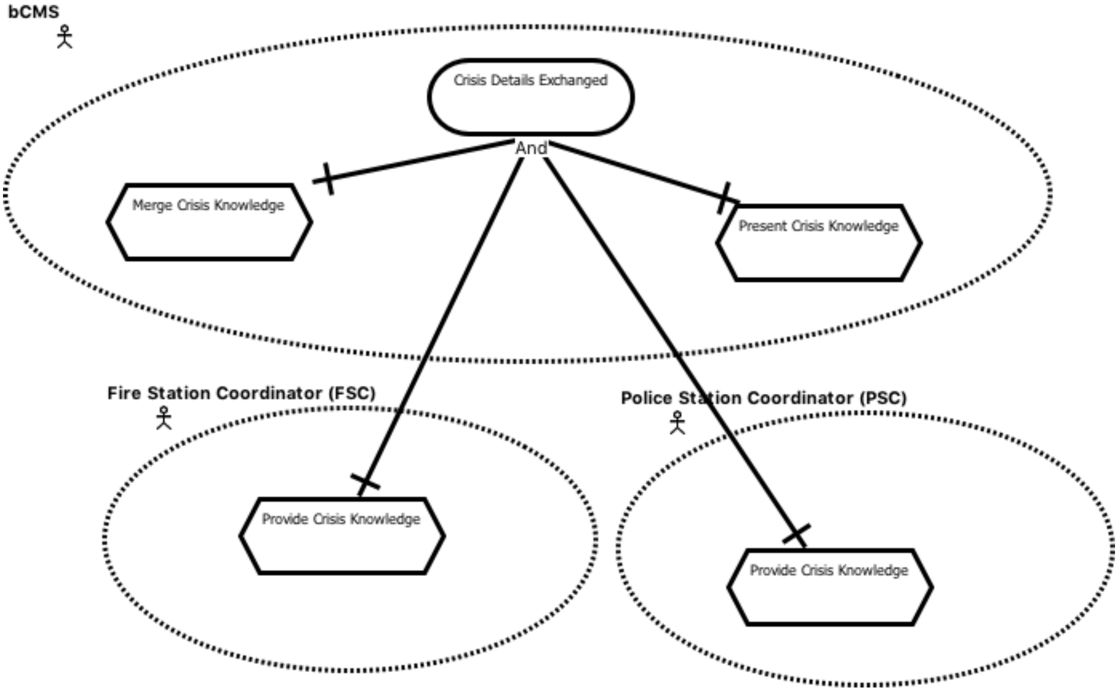


Figure 3.61.: Decomposition of `Crisis Details Exchanged`

On the basis of the knowledge about the crisis, the central goal of the coordination process, the coordination of the route plan, needs to be satisfied by the collaborating actors. Figure 3.62 provides a top-level decomposition of this goal. Vehicle counts need to be specified by each coordinator and communicated to the other. The `police station coordinator` then has to define a route for the police vehicles, which is to be acknowledged by the fire station coordinator. The route for the fire trucks is proposed also by the `police station coordinator`, but is not recorded unless the `fire station coordinator` agrees.

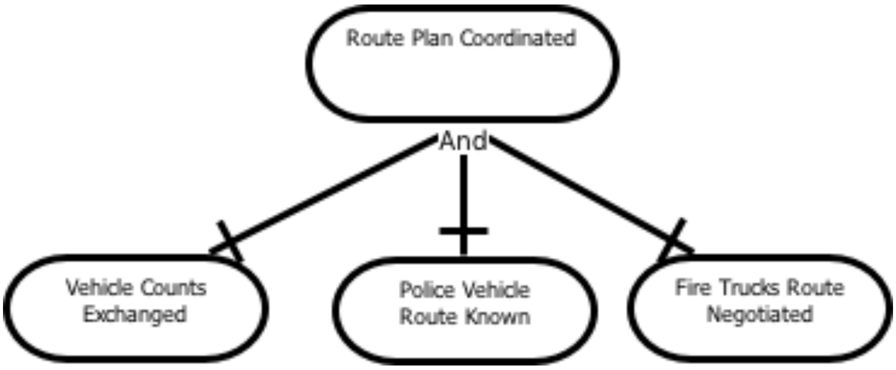


Figure 3.62.: Decomposition of `Route Plan Coordinated`

Vehicle Counts Exchanged is decomposed into the tasks for bCMS to record the provided vehicle counts of both coordinators and for each coordinator to provide the respective needed vehicle count (Fig. 3.63).

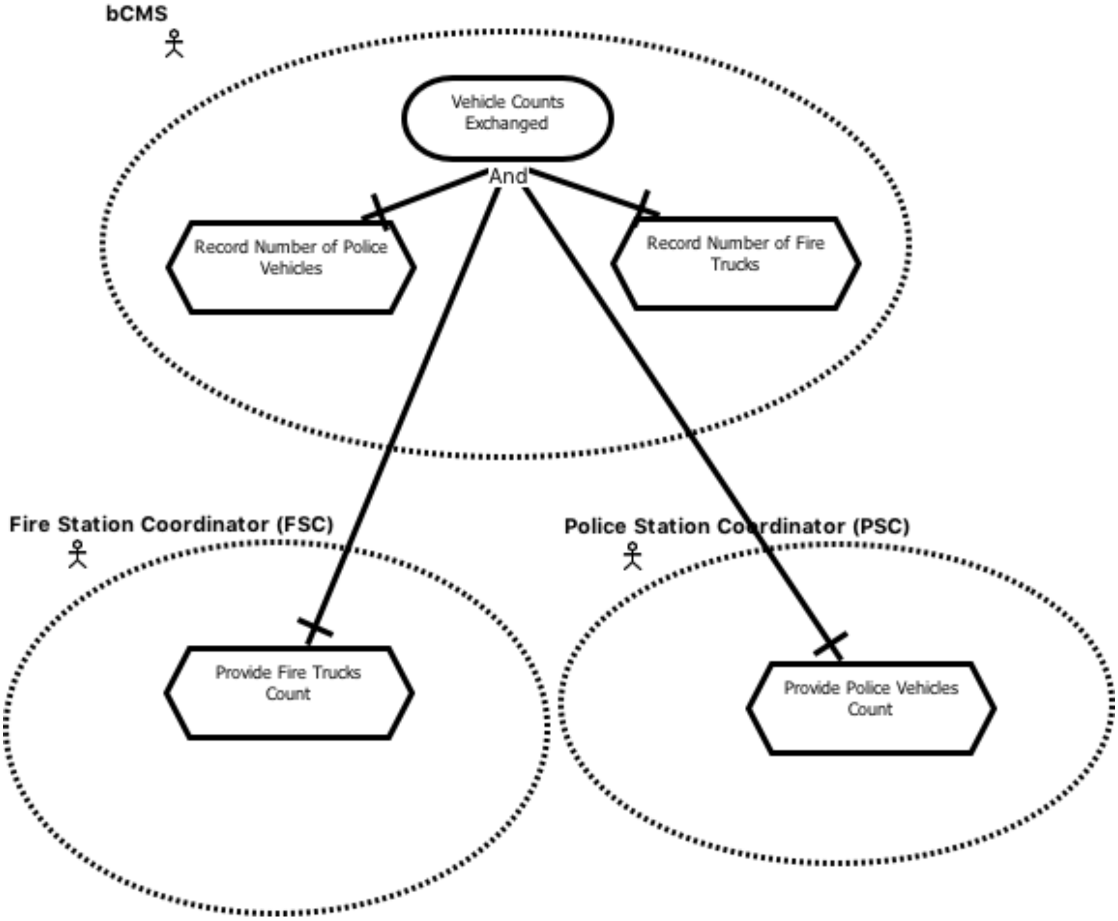

bCMS
Vehicle Counts Exchanged
And
Record Number of Police Vehicles
Record Number of Fire Trucks
Fire Station Coordinator (FSC)
Police Station Coordinator (PSC)
Provide Fire Trucks Count
Provide Police Vehicles Count

Figure 3.63.: Decomposition of Vehicle Counts Exchanged

For making the police vehicle route known to the fire station coordinator, bCMS has to record the route provided by the police station coordinator and present it to the fire station coordinator (Fig. 3.64).

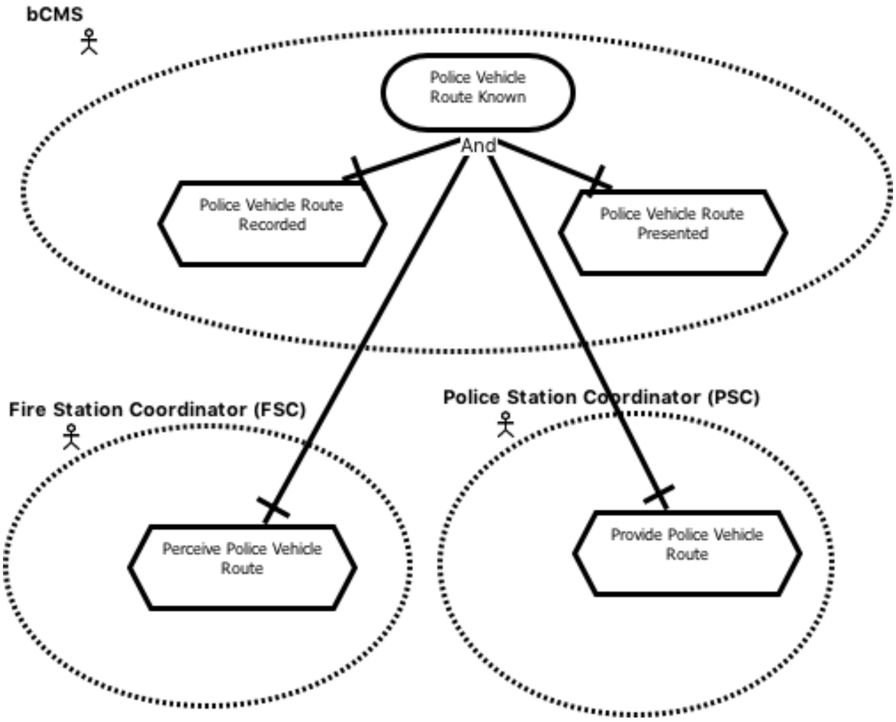

bCMS
Police Vehicle Route Known
And
Police Vehicle Route Recorded
Police Vehicle Route Presented
Fire Station Coordinator (FSC)
Police Station Coordinator (PSC)
Perceive Police Vehicle Route
Provide Police Vehicle Route

Figure 3.64.: Decomposition of `Police Vehicle Route Known`

The negotiation of the fire trucks route is a lot more complex, as the `fire station coordinator`'s consent is required. `bCMS` has to record the proposed route if it has been confirmed by the fire station coordinator, but if the `fire station coordinator` declines, the `police station coordinator` is required to provide an alternative route. In the extreme case, all proposed routes are declined, and the `police station coordinator` has to cancel the fire truck route coordination.

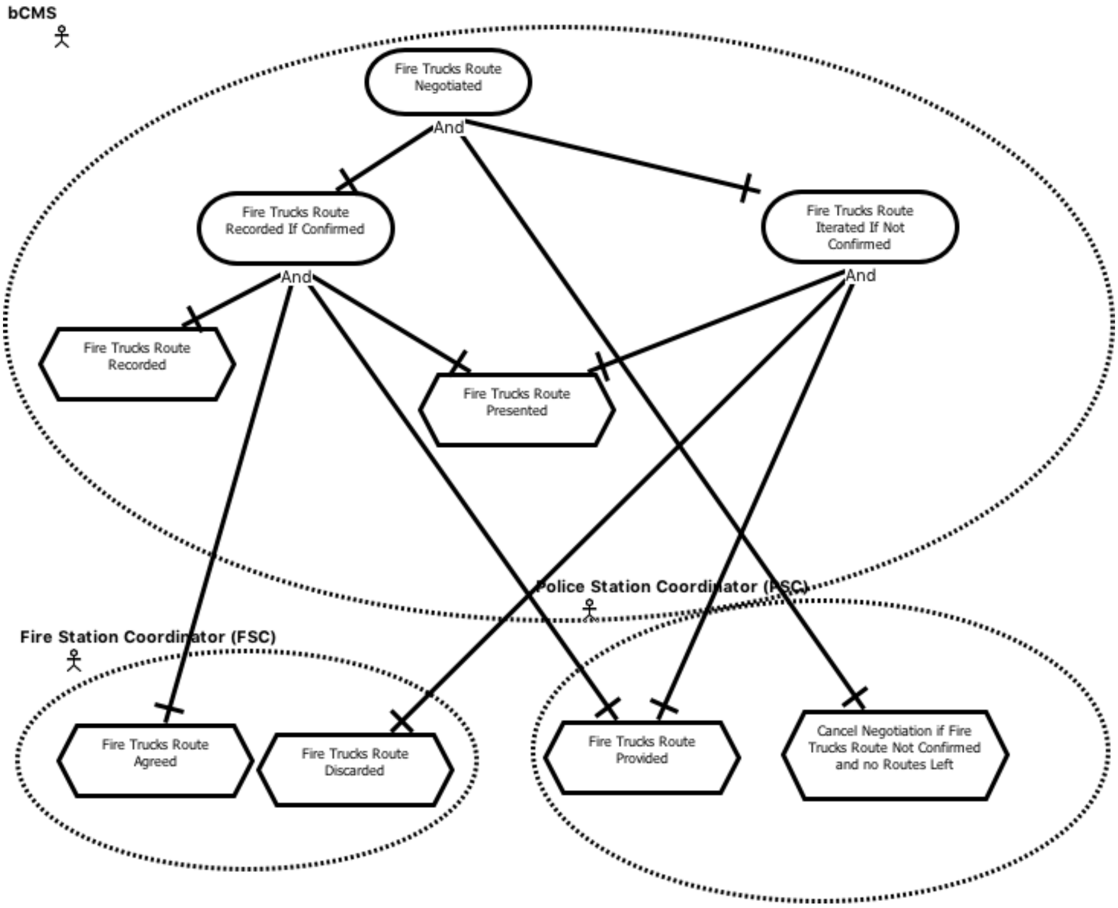

bCMS
Fire Trucks Route Negotiated
And
Fire Trucks Route Recorded If Confirmed
And
Fire Trucks Route Iterated If Not Confirmed
And
Fire Trucks Route Recorded
Fire Trucks Route Presented
Police Station Coordinator (PSC)
Fire Station Coordinator (FSC)
Fire Trucks Route Agreed
Fire Trucks Route Discarded
Fire Trucks Route Provided
Cancel Negotiation if Fire Trucks Route Not Confirmed and no Routes Left

Figure 3.65.: Decomposition of `Fire Trucks Route Negotiated`

Figure 3.66, an extended version of 3.61, shows a variant of the system in which the soft goal is to be satisfied that there is no unauthorized access of third parties to coordination details. The decomposition of `Crisis Details exchanged` is therefore extended to have another task, `Encrypt Crisis Knowledge`. A contribution link between `Encrypt Crisis Knowledge` and `Unauthorized Access to Coordination Details Prevented` shows the positive contribution of the encryption task to the higher-level soft goal. The positive contribution is expressed by the qualitative contribution type `Some Positive`, accompanied by an equivalent quantitative value `50`.

URN has only limited support for modeling variability. Figure 3.66 needs to be extended to also show the other system variant, in which the crisis details are exchanged unencrypted. To achieve this, the decomposition type of `Crisis Details Exchanged` would be changed to `XOR`, and two subgoals `Crisis Details Exchanged Unencrypted` and `Crisis Details Exchanged Encrypted` would be introduced. Both would refer to the respective tasks of `bCMS` and the coordinators. The existing contribution link could be moved to link the subgoal with encryption, and another contribution link with negative contribution would be added between the soft goal and the subgoal without encryption. As a result of the aforementioned modifications, the diagram would increase significantly in complexity. Furthermore, even though the contribution links might suggest the better alternative, there is no clear indication which alternative is currently chosen for the implementation of the system.

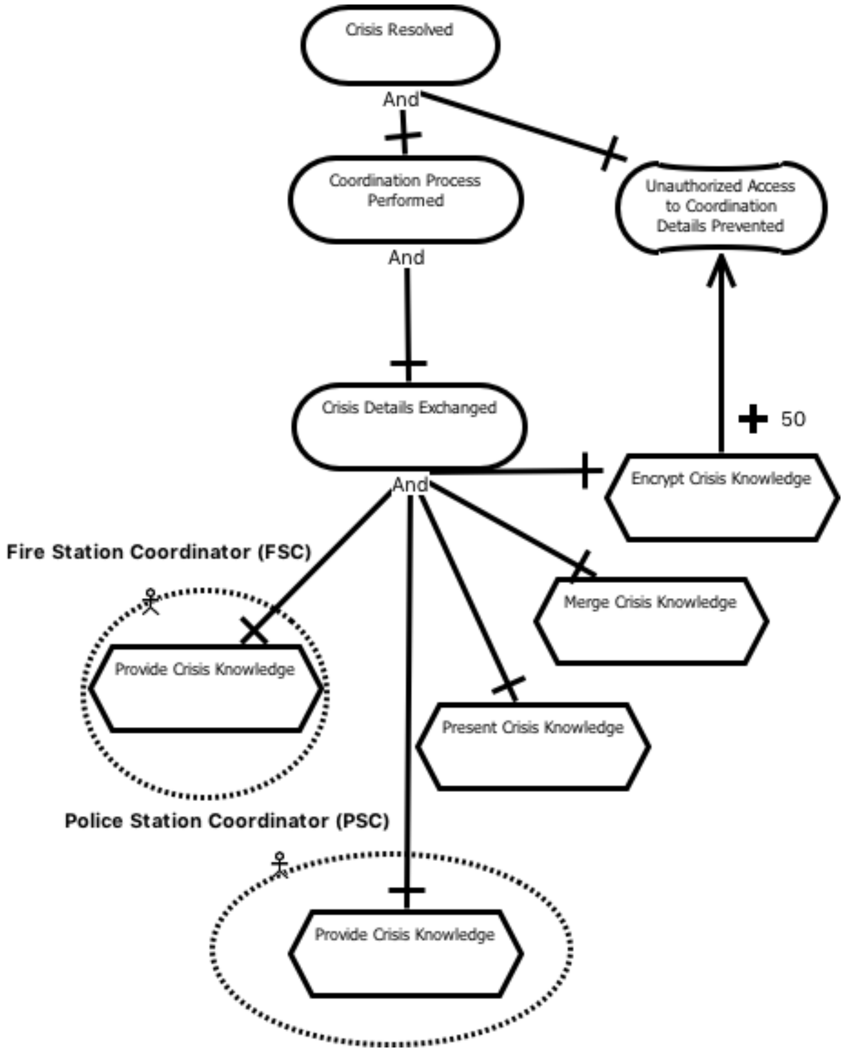


Figure 3.66.: One application of the variation point encrypted communication

Figure 3.67 provides another example of a contribution link. The figure addresses one "exceptional scenario" of the requirements document: `bCMS` shall record timeouts if the route coordination "exceeds a predefined limit". The requirement document does not motivate that scenario, thus the model in the figure is constituting a guess of the requirements analyst regarding that missing motivation. It is assumed that the average coordination time over multiple coordination sessions will decrease with the timeouts, as the timout logging will remind the coordinators in case they are taking too much time until providing certain information.

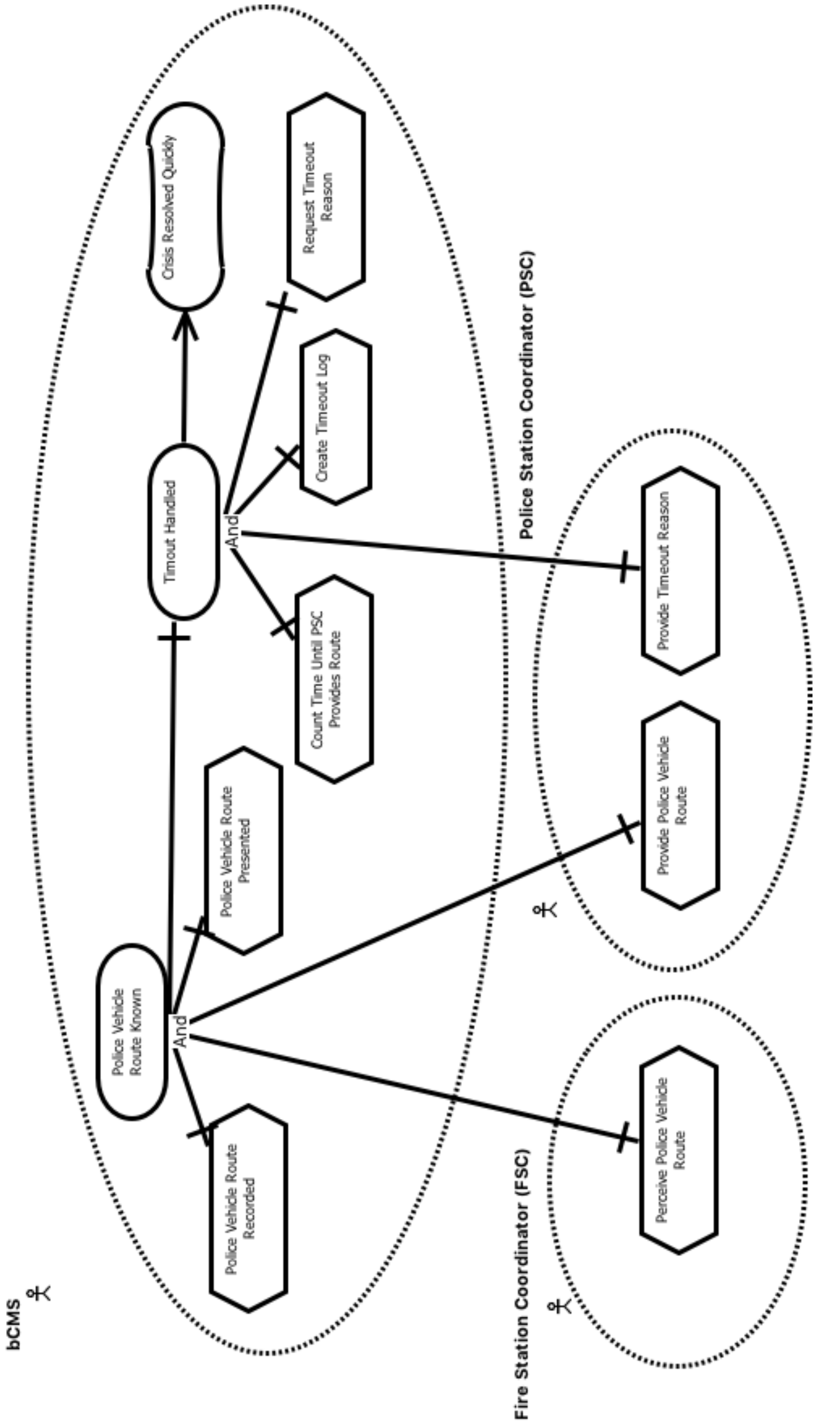

bCMS
Crisis Resolved Quickly
Request Timeout Reason
Create Timeout Log
Timout Handled
And
Count Time Until PSC Provides Route
Police Vehicle Route Presented
Police Vehicle Route Known
And
Police Vehicle Route Recorded
Police Station Coordinator (PSC)
Provide Timeout Reason
Provide Police Vehicle Route
Fire Station Coordinator (FSC)
Perceive Police Vehicle Route

Figure 3.67.: Modeling an extension point for timeout logs in terms of goals

As that assumption is still speculative, the contribution link has no quantitative or qualitative values yet.

While the respective parts of the GRL mode were presented in a certain order in this section, the model itself gives no indication about the ordering of activities to be performed by bCMS and its actors. On the following pages, some excerpts of a UCM model are provided to present information about the high-level interaction of the PSC and FSC actors with bCMS.
Figure 3.68 is a use case map of the initiation of the coordination session. Both coordinators connect in parallel. After both are connected, bCMS creates a session and for each of the coordinators requests the credentials. Each coordinator then provides credentials to be verified by bCMS. This use case map contains no conditional or exceptional behavior.

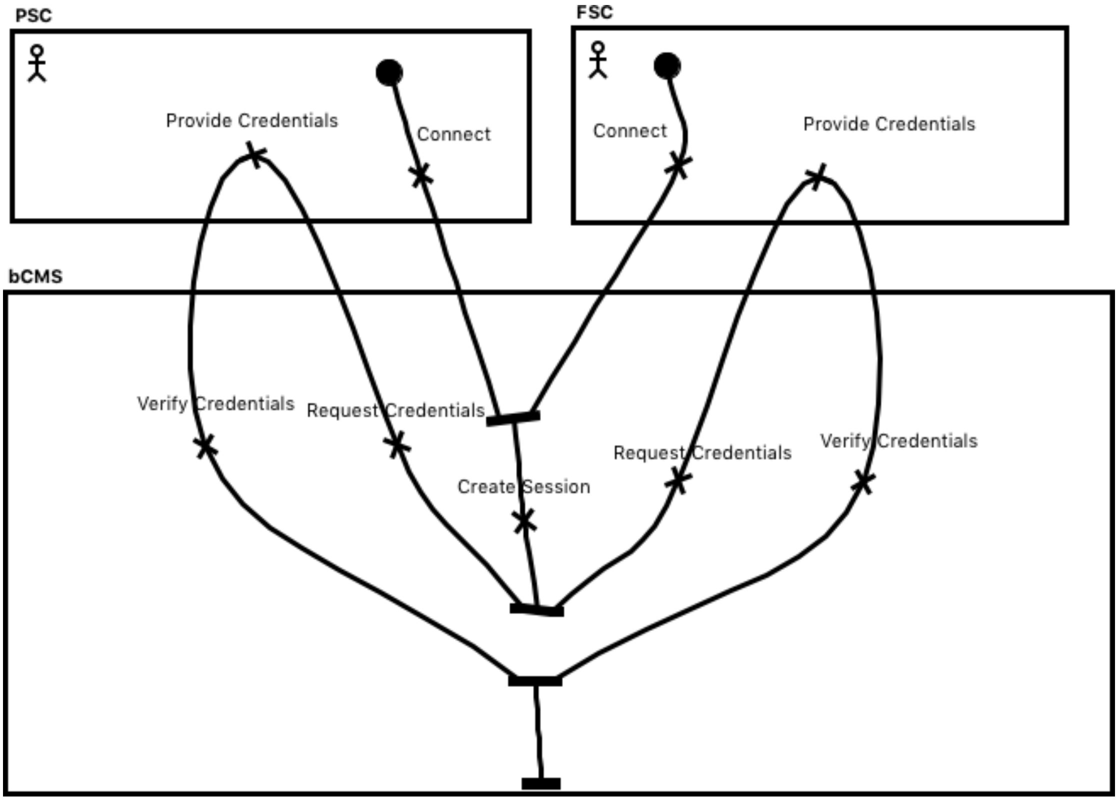


Figure 3.68.: Responsibilities upon session initiation

Figure 3.69 contains multiple scenarios regarding the goal `Police Vehicle Route Known`. The use case map contains some blocking and conditional behavior. The starting point is shown with an informal precondition stating that it is assumed that both coordinators have already exchanged the respective number of needed vehicles. The path is then forked into a branch on which the `FSC` waits until a route has been provided by the `PSC` and a path on which the `PSC` performs his task. To make the `PSC` aware of the task, bCMS requests provision of the route first. Then it starts counting the time until the PSC responds. If a predefined timeout interval is exceeded, the path branches into a timeout path on which bCMS creates a timeout log and requests a reason for the timeout from the PSC. If the PSC responds in time, the path continues normally. bCMS then records the police vehicle route. The waiting path, on which a waiting indicator has been displayed to the FSC, can now be continued, with the display of the route to the FSC, who has to inspect the route. After these steps, the negotiation of the fire trucks route can start.

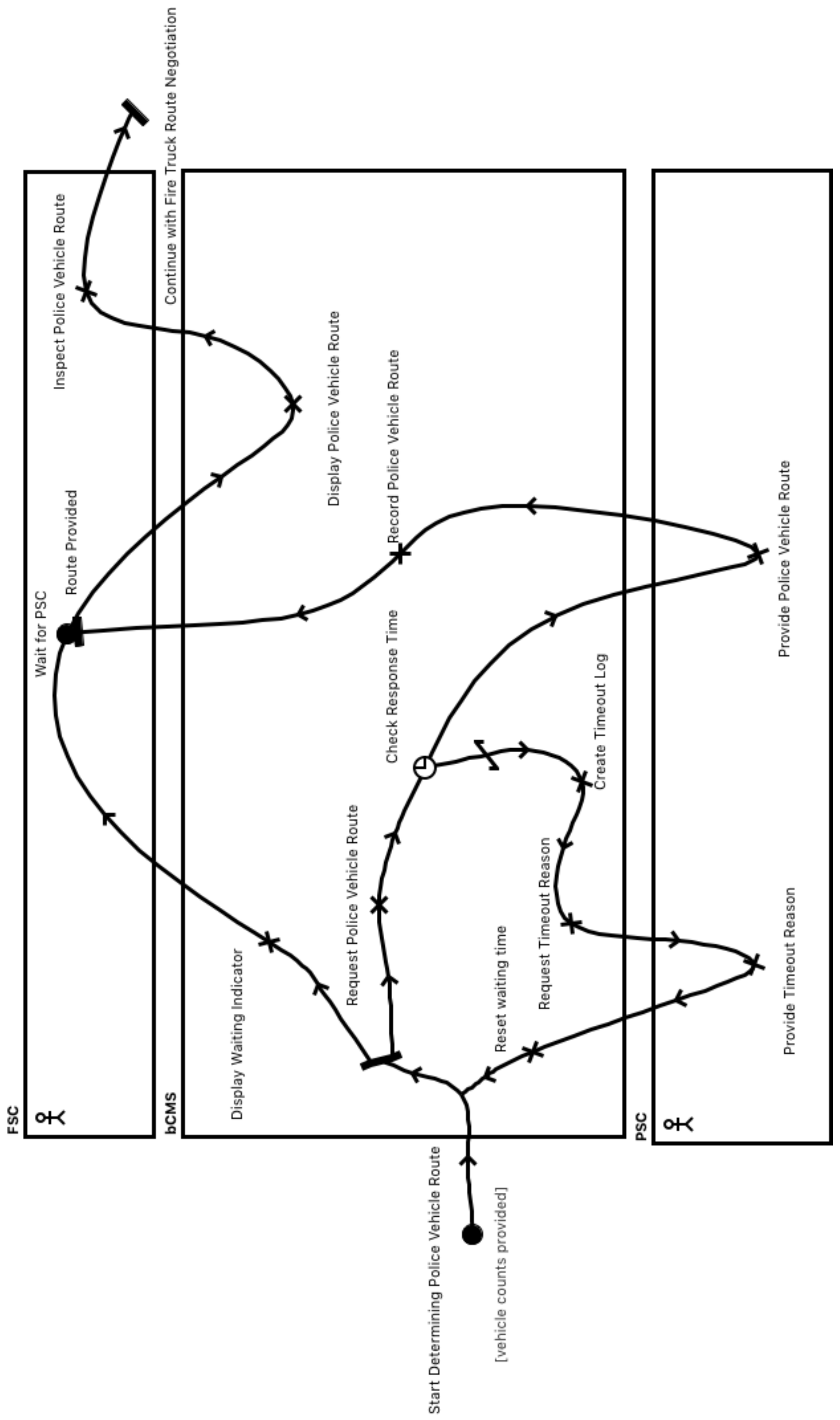

FSC
Wait for PSC
Route Provided
Inspect Police Vehicle Route
Continue with Fire Truck Route Negotiation
bCMS
Display Waiting Indicator
Display Police Vehicle Route
Request Police Vehicle Route
Record Police Vehicle Route
Check Response Time
Reset waiting time
Request Timeout Reason
Create Timeout Log
PSC
Provide Timeout Reason
Provide Police Vehicle Route
Start Determining Police Vehicle Route
[vehicle counts provided]

Figure 3.69.: Use case map describing scenarios for police vehicle route determination

3.2.4. Discussion

At first sight, it is surprising that the User Requirements Notation does not include any abstraction named requirement. The language user thus is required to have a good understanding about which URN abstractions imply requirements. This could be defined more clearly in the specification. While it states that GRL models focus on "non-functional requirements and quality attributes", it is unclear which abstractions support that focus. One of the central abstractions of GRL, `IntentionalElement`, has an attribute of type `IntentionalElementType`, whose enumeration literals bear very different meanings. `SoftGoal` clearly relates to non-functional requirements. `Goal` can relate also to functional requirements. `Belief` and `Resource` can only indirectly be related to requirements, and `Task` seems to focus on functionality.

Opposed to GRL, the URN specification states that UCM models focus on functional requirements, but also on operational requirements[10] and on "performance and architectural reasoning". At least the performance reasoning is related to performance (i.e. non-functional) requirements. Depending on how operational requirements are actually defined, they might or might not be related to non-functional requirements. Again, is unclear which abstractions should be understood as functional requirements. Most likely, the `Responsibility` abstraction has such a meaning, but it is not explicitly discussed in the specification.

The language requirements for URN have been specified in a separate ITU recommendation (ITU 2003). That recommendation already demands the partitioning of the language into two sublanguages, called URN-FR and URN-NFR. URN-FR shall focus on functional, and URN-NFR shall focus on non-functional requirements. In the language specification it is stated that GRL addresses the requirements for URN-NFR, and that UCM addresses the requirements for URN-FR.

Apart from the contradictoriness of the defined purposes of the GRL and UCM parts of URN, the loose coupling between these two sub-languages is a point of criticism. By linking GRL with UCM, URN would facilitate reasoning about how stakeholder intentions influence system architecture. But at first sight, there is no connection between UCM and GRL constructs. Looking a little closer reveals the `URNlink` class, that allows to link any UCM element with another. This generic link without predefined semantics is the only integration between UCM and GRL that URN currently has to offer. While it may not be a core requirement of a language specification, we were missing a section providing guidance on mechanisms of URN to facilitate the transition of early requirements knowledge modeled with GRL to late requirements knowledge modeled with UCM.

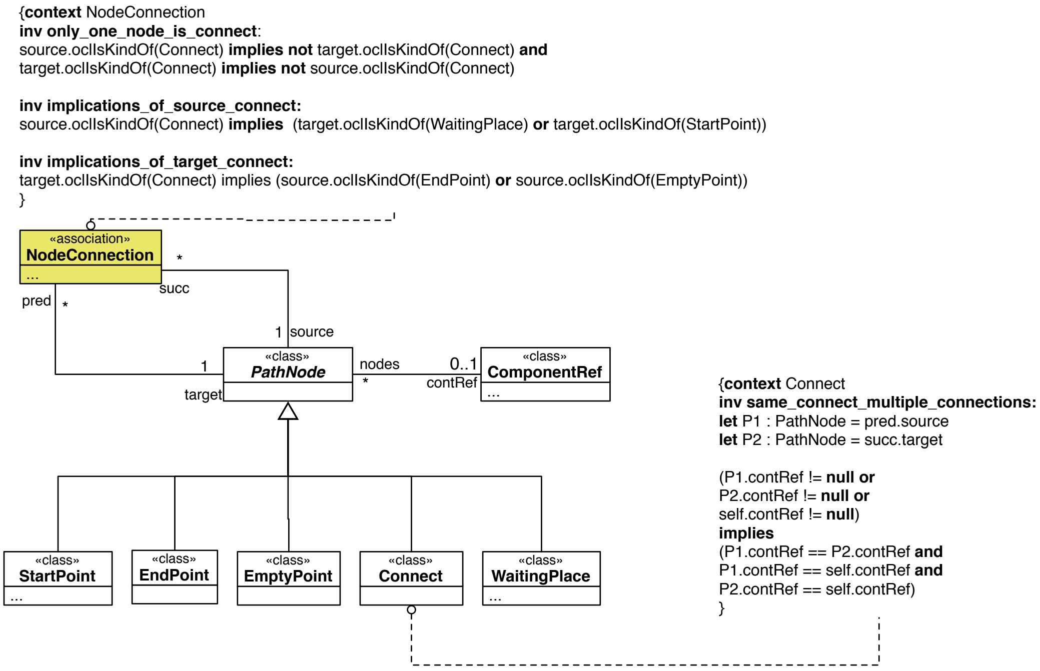
In general, the abstract syntax meta-model of URN is heavily relying upon constraints, and there is no rationale given why keeping the meta-model structure simple was preferred over minimizing the number of constraints. On some occasions, the possible instantiations of the meta-model are so constrained that the question arises whether adding additional structuring meta-classes would not have been more intuitive to under-

[10] The term "operational requirement" neither being defined in the language's requirements (ITU 2003), nor in the language's specifications (ITU 2008a; 2012).

stand. As an illustration, we will discuss two parts of the meta-model, one from GRL and one from UCM.

The first example are `IntentionalElements` of type `Belief`. On the syntax level, GRL knows only `IntentionalElement` as a meta-class, but not `Goal`, `SoftGoal`, `Task`, `Resource`, or `Belief`. These are enumeration literals of the `IntentionalElementType` enumeration. While the concepts certainly are different, it seemed to be out of scope to explicitly express their differences in terms of meta-classes. Some differences are only established by constraints on the relationship types of GRL. `Beliefs` cannot be decomposed or be part of a composition. A `Belief` cannot depend on something and nothing can depend on a `Belief`. Nothing can contribute to `Beliefs`. The only relationship that a `Belief` can participate in is when it is the source of a `Contribution`.

The second example is the `Connect` subclass of `PathNode` and its relationship to `NodeConnection`. From `PathNode` `Connect` inherits participation in the `NodeConnection` as source or target. But in fact, it may not. The multiple constraints shown in Figure 3.70, allow it to be part of a `NodeConnection` only under very special circumstances. For example, if the `Connect` is the `source` of a `NodeConnection`, that `NodeConnection's` `target` must not be a `Connect`, but a `WaitingPlace` or a `StartPoint`.



```
{context NodeConnection
inv only_one_node_is_connect:
source.oclIsKindOf(Connect) implies not target.oclIsKindOf(Connect) and
target.oclIsKindOf(Connect) implies not source.oclIsKindOf(Connect)

inv implications_of_source_connect:
source.oclIsKindOf(Connect) implies  (target.oclIsKindOf(WaitingPlace) or target.oclIsKindOf(StartPoint))

inv implications_of_target_connect:
target.oclIsKindOf(Connect) implies (source.oclIsKindOf(EndPoint) or source.oclIsKindOf(EmptyPoint))
}
```

```
{context Connect
inv same_connect_multiple_connections:
let P1 : PathNode = pred.source
let P2 : PathNode = succ.target

(P1.contRef != null or
P2.contRef != null or
self.contRef != null)
implies
(P1.contRef == P2.contRef and
P1.contRef == self.contRef and
P2.contRef == self.contRef)
}
```

Figure 3.70.: Constraints related to `Connect`. Adapted from section 8.2.18 of the URN specification (ITU 2012)

It might be argued that the average language user will never experience the language in terms of its meta-model. However, if the meta-model had an intuitive structure, it could be the basis of learning and teaching the language. To hide the complexity of the

meta-model, teachers will have to come up with a slightly different terminology or will have to live with the fact that their teaching might be in parts contradictory to the specified meta-model. As an example: the teacher decides to introduce the term "intentional element" and then states "intentional elements can be goals, soft goals, tasks, resources, or beliefs". He will have trouble then stating that "dependencies are relationships between intentional elements", because that is partially wrong. He would either have to state "dependencies are relationships between intentional elements, but not beliefs" or "dependencies are relationships between goals, soft goals, tasks, or resources". Both statements are vague and irritating because they suggest that there is some commonality between the `IntentionalElementTypes` `Goal`, `SoftGoal`, `Task` and `Resource`, but a difference to `Belief`, but there is no terminology from within the language to speak about it. It seems like a category is missing. This problem will also occur, if the teacher decides not to speak about `IntentionalElement` at all, but only about its subtypes.
It is also unusual that in the graphical representation of the meta-model, which seems to be UML, abstract classes are not represented with class names in italics. Whether a meta-class is abstract is only specified in its constraints, which is hard to follow especially if the constraints are inherited from a superclass.
The complexity of the meta-model also affects its implementability. The various constraints all have to be implemented, which will force the implementors to introduce their own taxonomy or use various conditional statements. In the domain of software design, there is as a bad code smell called "Long Method" (Fowler and Beck 1999), of which long conditional statements are a contributing factor. That aspect of source code complexity can be resolved by a refactoring called "Replace Conditional with Polymorphism".
Finally, the complexity of the meta-model will also affect users of a tool implementation of URN. A tool implementation correctly implementing the constraints will keep a user from connecting `IntentionalElements` of type `Belief` to some differently typed `IntentionalElement` with `Dependency` or `Decomposition` links. This might be easy to learn, but that `Contribution` links only work in one direction for `Belief` will be at least irritating in the first place.

As all the meta-model constraints of URN are specified in natural language, tool conformance to URN will be hard to prove. As URN's interchange format also does not enforce the constraints, different tool implementations might end up exchanging incompatible URN models. Another general observation regarding URN's meta-model is only almost MOF-compliant. It is violating EMOF constraint number [2][11] (OMG 2015b), which states that "the option is disallowed of suppressing navigation arrows such that bidirectional associations are indistinguishable from non-navigable associations". The constraint is violated for various associations, e.g. the associations between `NodeConnection` and `PathNode` in Figure 3.51.

According to Moody et al. (Moody, Heymans, and Matulevičius 2009) and Genon et al (Genon, Amyot, and Heymans 2011), it can be stated that the notation of URN has not been designed with Moody's principles in mind. The *perceptual discriminability* of several symbols is low, for example the symbols representing `IntentionalElements`

[11] Or CMOF constraint number [5], which is the same.

and Indicators or the symbols for ComponentRefs. The IntentionalElement symbols except the one for Belief basically are variations of a rectangular shape and by default have the same width and height. The Belief symbol - an ellipse - is close to the Goal symbol as the high corner radius of the Goal shape makes it somewhat similar to an ellipse. The Indicator symbol has the same shape as the Task symbol. We have also observed *symbol excess*, i.e. non-optimal *semiotic clarity*. For example, the representation of XOR/IOR decomposition has a redundant representation called "means-end" notation. The *semantic transparency* of the symbols is low, as most of them are based on abstract shapes, from which the underlying abstraction's meaning cannot be easily derived. Decomposition "arrows" are not very intuitive to read as they do not show direction.

3.3. Very Lightweight Requirements Modeling Language (VLML)

Glinz and Wüest discuss the requirements of an "Ultralightweight Requirements Modeling Language (ULM)" and sketch a language to implement these requirements (Glinz and Wüest 2010)[12]. In a subsequent position paper, Glinz names that sketched "concrete ULM" a "Very Lightweight Requirements Modeling Language (VLML)" (Glinz 2010). For ease of reading, we will refer to the concrete language in all following paragraphs as VLML, while the general vision of a lightweight modeling language as conceived by Glinz and Wüest is still called ULM.

Glinz and Wüest observe that "natural language is still the dominant language for writing requirements specifications in practice", where natural language expressions are "augmented with tables, pictures, and [...] some isolated model diagrams"(Glinz and Wüest 2010, p. 3). They give four reasons for the fact that natural language still dominates: The first reason is "the inability of industry to adopt the existing modeling technology". The second reason is that "heavyweight modeling languages such as UML don't fit the needs of industrial requirements engineers". A third reason is that the natural language-dominated style "outperforms today's requirements modeling languages". They go even further to claim that this was shown in a contest held to compare modeling languages (See (Gotel and Cleland-Huang 2009) for details of the contest), where "the combined power of natural language and [...] pictures outperformed all modeling approaches". The fourth reason is that "non-functional requirements [...] cannot be expressed as models".

Glinz and Wüest propose the "creation [...] of an ultralightweight modeling language" (ULM) to overcome the dominance of natural language approaches to requirements. They criticize natural language specifications as unstructured and informal. They argue that in between "heavyweight" modeling languages and natural-language requirements specifications, there is still room for a new kind of language: An ultra-lightweight requirements modeling language that combines the textual nature of requirements with the narrative and pictorial nature of information gathered in early stages of the requirements

[12] If not otherwise specified, we are citing from this work in this section.

elicitation process.

They specify seven broad requirements for a ULM[13]:

R1. A ULM shall provide strong support for writing textual requirements and drawing pictures.

R2. A ULM shall provide model elements for structuring text.

R3. A ULM shall provide model elements for drawing pictures as diagrams, with individually identifiable elements and some very lightweight semantics.

R4. A ULM shall be visual in the large and textual in the small.

R5. A ULM shall enable the creation of tools that provide powerful means for editing, navigation, selective visualization and analysis of specifications.

R6. A ULM shall be easy to read, to write and to learn.

R7. The visual syntax of a ULM shall be well-designed with respect to the design principles for visual notations.

In addition to the seven requirements towards an ULM, Glinz and Wüest specify a set of "design considerations" for VLML [14]. The first consideration, "*hierarchical structure*" deals with **R5**: for the visualization of hierarchical structure, nested boxes are preferred over graphic or linear trees. This structuring mechanism is borrowed from ADORA (Glinz, Berner, and Joos 2002; Reinhard et al. 2008).

The second consideration, "*contextualization*", deals with the language providing context for every model element through the specification of paths. The context of any element can be specified by a fully qualified name at the top of the box representing an object. It addresses parts of **R3** ("identifiable elements").

In order to support requirement **R6**, Glinz and Wüest discuss a "*small visual vocabulary*" as third consideration. The objects types of VLML are `Technical item`, `Living item`, `Connector`, and `Picture`. Objects can be connected by `static`, `influence`, `flow`, or `nesting` relationships. `Technical` and `Living Items` can be detailed with modifiers `External`, `Fuzzy`, `Multiple`, and `Boundary`. The modifiers are visualized by modification of the border of the box depicting a technical item, or as decoration to the living item symbol.

The fourth consideration, "*naming*", addresses **R2**. VLML is to support the promotion of text fragments to identifiers. Text fragments that are identifiers can provide links between model elements. Furthermore, identifiers can be used to annotate pictures that are part of the model. The latter aspect partially addresses **R1**, as the annotation of pictures with identifiers connects the textual requirements with the drawn pictures. It thus also is a part of addressing **R4**.

[13] These requirements are literally cited from (Glinz and Wüest 2010), quotation marks were omitted.

[14] It is not clearly stated which design consideration relates to which requirements. The connections we draw here are our own analysis.

"*Semantic enrichment*", the fifth consideration cannot be traced to a particular requirement mentioned above. It is about keeping the language "lightweight". In our point of view, the arguments is that while the "small visual vocabulary" makes the language easy to learn, it makes it less expressive. To compensate for that weakness, VLML can incorporate enrichments, that are concepts borrowed from other languages. The benefit of enrichments is that they can provide an "evolution path" to models expressed in other languages. These then more heavyweight models would be used in later phases of the requirements engineering process. Enrichments can be syntactical by introducing new abstractions, or modifications of semantics of existing VLML abstractions. Glinz and Wüest propose two standard enrichments that modify semantics of connectors: `AND` and `OR` connectors.

The sixth and last design consideration is "*lightweight analysis*", refers to **R5** and concentrates on the needed tool support for VLML. The hierarchical visualization mechanism needs to be supported as well as the naming and picture annotation mechanism. In addition, a VLML tool shall support a set of automated validation rules. A VLML model accompanied by a tool to perform the validation has an advantage over specifications that use unrestricted natural language.

An interesting aspect of VLML is neither mentioned in the requirements nor in the design considerations. VLML allows a visual expression of where the model has incomplete information: "Incompletely specified objects are marked with an ellipsis after their names". Also interesting are the mechanisms to provide complexity management: If parts of a model element are omitted from a diagram, three dots can be added to its name as a suffix. If cardinalities and relationship names are suppressed on a diagram, the relationship line is to be decorated by a "pull down handle symbol". For object hierarchies, any part of the hierarchy can be collapsed and expanded again. As soon as the first part of an object gets collapsed, its name is decorated with the "three dots" notation.

Wüest and Glinz also describe *FlexiSketch*, a sketch recognition tool that allows to semi-automatically transform hand-drawn diagrams into models of an ULM(Wüest and Glinz 2011). We think that it is especially the small visual vocabulary that greatly simplifies the construction of such a sketch recognition tool. In addition, FlexiSketch adds an additional requirement to the notation of an ULM: The symbols of the notation need to be drawable. This is a refinement of the "easy to write" requirement. We assume that FlexiSketch also requires strict adherence to one of Moody's design principles, perceptual discriminability.

A new possibility for complexity management in graphical models that is different from the ADORA approach, but would also qualify for a tool support of an ULM, is described by Ghazi, Seyff and Glinz (Ghazi, Seyff, and Glinz 2015). It uses a spring and magnets metaphor to allow users to guide the behavior of layout algorithms, which are needed when parts of the model are collapsed and expanded.

3.4. Other Modeling Languages and Approaches

i* (Yu 1997) is a predecessor of URN's GRL sub-language, but is still referred to as a separate language in many scientific publications. For example, i* is the basis of the Tropos software development methodology (Castro, Kolp, and Mylopoulos 2001). As the elements of i* are largely the same as those of GRL (discussed in Section 3.2.1), it is not discussed any further in this section.

Bleistein, Cox, Verner, and Phalp describe the *B-SCP* framework, "a requirements analysis framework for validating strategic alignment of organizational IT based on **s**trategy, **c**ontext, and **p**rocess"(Bleistein et al. 2006)[15]. The main purpose of the framework is to align requirements to "organizational IT" with the "business strategy" of an organization. Business strategy is modeled with i* diagrams(Yu 1993), the business context with a subset of Jackson's problem diagrams (Jackson 2001), and the business process with role activity diagrams (Ould 1995). The contribution of B-SCP is that the three kinds of diagrams are not used unconnected, but are tightly integrated within the framework. For example, problem diagrams are re-defined in such a way that the requirement part of a problem diagram is now provided by an i* goal model.

Li et al. argue that goal-oriented requirements approaches have been too orthodox in interpreting non-functional requirements as soft goals but should instead allow to interpret some of them as "requirements for qualities"(Li et al. 2013). They sketch an abstract syntax of an *extension to i** to incorporate that focus change.

Lee et al. propose a *"goal- driven feature modeling approach"* (Lee et al. 2013). Though the first publication on feature modeling, the report on FODA (Kang et al. 1990), does explicitly mention a relationship of feature modeling to requirements elicitation, Lee et al acknowledge that in the field, feature modeling has been considered as a software product line engineering (SPLE) technique and not a requirements engineering technique. From the opposite viewpoint, RE techniques weren't considered as adequate for contributing to SPLE. Lee et al. therefore see a need for reconciliation. The proposed approach partitions a feature model into two categories of viewpoints: *problem space* viewpoints are *goals*, *qualities*, and *usage context*, whereas *solution space* viewpoints are *capabilities*, *operating environments*, and *design decisions*. A major criticism is that every viewpoint is considered as a feature model, and thus every element of a viewpoint a feature. We do not see the benefit of re-interpreting e.g. goals or usage context as features. However, we share the observation that a requirements modeling technique to support product line engineering should consider feature variability and its implications on requirements as an important concern. While feature models alone show only the variability of features, an integrated model could analyze the effects of that variability on the requirements, and thus show requirements variability (or goal variability, or any other kind of requirements related knowledge influenced by feature choices).

As pointed out by Rolland and Salinesi, there is a long tradition of works that discuss the relationship of goals to scenarios or use cases (Rolland and Salinesi 2009). Rolland

[15] Emphasis in citation added by the author, to illustrate that B-SCP is an acronym for **b**usiness, **s**trategy, **c**ontext, and **p**rocess

and Salinesi describe one of them, the *L'Ecritoire* approach, to point out how scenarios can drive the discovery of goals. It combines goals and scenarios in abstractions called *requirement chunk*. L'Ecritoire was first described by Rolland et al. (Rolland, Souveyet, and Achour 1998) and as a part of the CREWS research project (Jarke 1999). It has also been incorporated into the RESCUE requirements engineering process (Maiden and Jones 2004).

The goal of *UML (Unified Modeling Language)* (OMG 2011c) is to "provide system architects, software engineers, and software developers with tools for analysis, design, and implementation of software-based systems as well as for modeling business and similar processes" (OMG 2011c). It (OMG 2014a) settled the so called "method war" that took place between nine competing approaches to object-oriented analysis and design of software (Fowler 2004, 5-6). While the UML entails analysis, UML is design-centric, focusing on the design and implementation of systems. It can particularly be used to describe the technical details of a system under discussion in terms of structure and behavior. UML is not a requirements modeling language because it does not support the modeling of requirements beyond use cases. Use cases model high-level functional requirements: A use case tells what kind of function is required from a system in order to be useful for a certain kind of user. Apart from that, "UML offers little for documenting, modeling, and structuring functional requirements" (Broy 2010). Nine more deficiencies of UML with respect to requirements engineering are identified by Glinz(Glinz 2000). In the Unified Software Development Process (Jacobson, Booch, and Rumbaugh 1999), a use-case driven analysis also entails identifying boundary, entity and control classes. These classes were, as stereotypes, a part of a standard profile to extend UML, called "UML Profile for Software Development Processes" (OMG 2000). But since UML 2.0, that profile is not part of the UML specification anymore.

SysML (Systems Modeling Language) (OMG 2012) is an extension of UML that has notational support for expressing requirements. SysML adds a concept for representing requirements in graphical models, thus offering potential traceability between SysML diagrams and requirements documents. It does, however, not allow to clearly delineate functional from non-functional requirements. Its `Requirement` abstraction can be used to model both: "A requirement may specify a function that a system must perform or a performance condition that a system must satisfy" (p.147). There is only a weak relationship between SysML's `Requirement` and `UseCase` abstractions. The generic relationship `Refine` can relate any SysML element with a requirement. The SysmL specification is ambiguous on this matter. It suggests using a "use case or activity diagram" for refining requirements but in an example diagram, the `Refine` relationship is between a `UseCase` instance and a `Requirement` instance. Beyond `UseCases` and `Requirements`, SysML has no support for early requirements modeling.

Glinz et al. have developed *ADORA* (**A**nalysis and **D**escription **o**f **R**equirements and **A**rchitecture), an object-oriented language "to be used primarily for requirements specification and also for logical-level architectural design"(Glinz, Berner, and Joos 2002; Reinhard et al. 2008). The authors claim to overcome some of UML's deficiencies regarding requirements specification as analyzed by Glinz (Glinz 2000). ADORA models are constructive models, which means that a model expressed in ADORA constitutes an abstract model of the future system.

The central abstraction of the language is the `Abstract Object`. An abstract object inserts a meta-level between the abstractions class and object known from UML (See Figure 3.71).

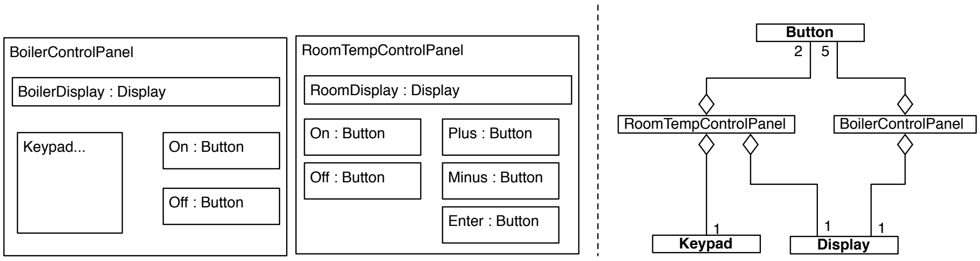


Figure 3.71.: Abstract Objects vs. Classes. Adapted from (Glinz, Berner, and Joos 2002, p.489).

The structure of an abstract object is determined by its properties and its parts. The properties of an `Abstract Object` are its `Attributes`, `Relationships`, and `Operations`. The parts of an `Abstract Object` are either `Abstract Object`, `Abstract Object Sets`, `States`, or `Scenarios` (Glinz, Berner, and Joos 2002). With these abstractions, ADORA supports the modeling of structure, functionality, and behavior. The context of the system under discussion in terms of `Actors` and `External Objects` that are used to model COTS components.

A model in ADORA is strictly hierarchical, i.e. the path to any object in the model can be unambiguously be specified. ADORA supports complexity management via views instead of diagram types. To achieve coherency of the views, for a concrete view on the model a base view is combined with one to many aspect views. The base view reflects the basic structure of the system under discussion. Structural, behavioral, user, and context aspect views can be superimposed on that basic structure. In any view, details of an element may be suppressed. Information hidden from the current view is represented by an `is-partial` indicator, which adds three dots to the name of the `Abstract Object` for which information is suppressed from the view.

The behavior aspect of abstract objects is modeled visually with a notation that is similar to Harel state charts (Harel 1987). The functionality aspect of an `Abstract Object` is specified textually via specification of `Operations`. `Operations` are specified in a separate formal, axiomatic language called ADORA-FSL (Joos 2000). The user aspect is in our point of view also related to the functionality of the system. The user aspect focuses on the interactions of `Actors` with the system, including the `Scenarios` they are participating in and the `Abstract Objects` (or `- Sets`) they are interacting with. Complex scenario decompositions can be modeled visually with a scenariocharts, based upon Jackson's structure diagrams (Jackson 1983). The context aspect is redundant to the user aspect as it also adds `Actors` to the basic structure, but does not entail `Scenarios`. In addition to the `Actors` and their relationships to the system's objects,

the context aspect can also show External Objects.
ADORA has been extended to not only support partial views, but also partial models (Xia 2004). The semantics of the is-partial indicator has been extended so that it can also represent information that is not yet part of the model. This can help the requirements analyst to record the “known unknowns” (Sutcliffe and Sawyer 2013). Together with a mechanism to support concretization of partial models the extended ADORA provides an evolution path between different model versions that gradually become less abstract. A simulation engine was built to support simulation of partial scenario models (C Seybold, Meier, and Glinz 2004) to allow for gradual concretization. This enables a scenario-driven elicitation process (Christian Seybold, Meier, and Glinz 2006) with ADORA. The language has further been extended to support modeling of aspects (Meier et al. 2007) and variability (Stoiber, Meier, and Glinz 2007).
Compared to Glinz’ ULM vision discussed in section 3.3, ADORA, including the aforementioned extensions, is a heavyweight requirements modeling language to support all phases of the requirements engineering process. With its support for partial models, it can facilitate the transition between early and late requirements models. However, it is missing abstractions for describing goals, dangers, and stakeholders beyond actors.

3.5. Model-Based Requirements Acquisition Strategies

The “acquisition assistant” proposed by Dardenne et al. (Dardenne, Fickas, and Lamsweerde 1991; Dardenne, Lamsweerde, and Fickas 1993) is driven by a meta-model, which structures the “requirements database” and “requirements knowledge base”. The *requirements database* contains the requirements model of the system under discussion. The *requirements knowledge base* contains “*domain-level knowledge*”, concerning “concepts and requirements typically found in the application domain”, and “*meta-level knowledge*” concerning “properties of the abstractions found in the meta-model [..] and ways of conducting specific acquisition strategies”, including concrete acquisition tactics.
Dardenne et al. propose an acquisition strategy that is tailored to the KAOS meta-model presented in Section 3.1. It is called “*goal-directed requirements acquisition*”, and consists of seven parts (Dardenne, Lamsweerde, and Fickas 1993; Lamsweerde, Darimont, and Massonet 1995):

1. Acquisition of Goal structure and identification of concerned Objects.
2. Preliminary identification of potential Agents and their Capabilities.
3. Operationalization of Goals to Constraints.
4. Refinement of Objects and Actions.
5. Derivation of strengthened Actions and Objects to ensure Constraints.
6. Identification of alternative Responsibilities.
7. Assignment of Actions to responsible Agents.”

Tactics to support the goal-directed strategy where proposed by Dardenne et al. (Dardenne, Lamsweerde, and Fickas 1993), Darimont and van Lamsweerde (Darimont and Lamsweerde 1996), and Letier and van Lamsweerde (Lamsweerde and Letier 2000; Letier and Lamsweerde 2002). Exemplary tactics are "Reduce goals into subgoals so that the latter require cooperation of fewer potential agents to achieve them" (Dardenne, Lamsweerde, and Fickas 1993)or "Introduce Tracking Object" to resolve unmonitorability of an Object by an Agent (Letier and Lamsweerde 2002).

The idea that a meta-model drives the acquisition process is also found in the "*requirements apprentice*" (RA) described by Reubenstein and Waters (H. B. Reubenstein and Waters 1989; Reubenstein 1990; Howard B Reubenstein and Waters 1991). The requirements apprentice is an interactive system, in which requirements analysts can enter statements expressed in a special command language. The statements express what the requirements analyst already knows about the system under discussion. The RA system can analyze the statements in order to resolve ambiguity, contradiction, and incompleteness. The current state of knowledge about the system's requirements is held in the *requirements knowledge base*. Pre-existing, domain-specific requirements knowledge, is stored in a *cliché library* in the form of "*requirements clichés*" . Both requirements clichés and the command language are driven by a "requirements ontology" including the abstractions `Agent`, `Behavior`, `Action`, `Operation`, `Functional-Requirement`, `Need`, `Product-Function`, `Policy`, `Problem`, and `Solution`. The strategy proposed by Reubenstein et al. is not characterized by a specific order in which the ontology or the cliché library should be traversed. Instead, what could be called an *incremental informality reduction strategy* is driven by the idea that the requirements analyst can input statements in any order into the RA, with the tool giving assistance in gradually transforming informal statements into a formal requirements model.

Other authors have also proposed requirements acquisition strategies without calling them so. For example, Jacobson et al. characterize the analysis phase of the software development process as two interacting sub-activities (Jacobson et al. 1992, p.123). The outcome of the requirements analysis activity is the requirements model, consisting of `Use Cases`, `Actors`, a `System Boundary`, `Interface Descriptions`, `Problem Domain Objects`. The requirements model is refined by the analysis model in the robustness analysis activity. The resulting model is called the analysis model, including `Use Cases`, `Actors`, a `System Boundary`, `Boundary Objects`, `Entity Objects` and `Interface Objects`[16]. We call this a *use-case driven requirements acquisition strategy.*

The *L'Ecritoire* approach of Rolland et al. (Rolland, Souveyet, and Achour 1998) supports a *requirements acquisition strategy based on goal-scenario coupling.* The central abstraction of the underlying meta-model is the `Requirement Chunk`. It pairs a `Scenario` with a `Goal`, combining an intentional and an operational view. The meta-model is shaped by the duality of `Goal` and `Scenario`: Apart from the `Requirement Chunk`, it can be partitioned into a `Scenario-` and a `Goal-related` part. A `Scenario` is described by a set of `Actions`, which can be hierarchically decomposed according to the composite pattern (Gamma et al. 1995). `Atomic Actions` are related to cooperating

[16] The possible relationships between the abstractions of both models are omitted here for brevity.

Agents, and can have Objects as parameter. Objects are specialized by Agents or Resources. Each Scenario has an Initial- and a Final State.

The structure of a Goal is given by a Verb and a set of Parameters. Parameter is abstract, it is further specialized by the abstract classes Target, Direction, Way, and Beneficiary. The Target of a Goal is either an Object or a Result, it "designates entities affected by the goal". The Direction is either a Source or a Destination, it identifies "the initial and final location of objects to be communicated". The Way of a Goal is either described by Means or a Manner. Means describe instruments to be used to satisfy a Goal. A Manner "defines the way in which the goal is to be achieved". As this can be described in terms of Goals, Manner contributes to a recursive definition of Goals. A Beneficiary is a person or group of persons being positively affected by a Goal.

L'Ecritoire's meta-model facilitates three "discovery strategies" for the identification of new requirements knowledge. One strategy identifies new knowledge by refinement, one by composition, and one by variation. In the refinement strategy, Actions of the Scenario lead to identification of new Goals. The strategy consists of two "guiding rules". In the first rule, every Action of a Scenario is a candidate for identifying another sub-Goal that *refines* that Chunk's Goal. The second rule is based on an Action taxonomy that is not explicit in the meta-model. The idea is that there are three kinds of Action pairs, "information provision/request, service provision/request, condition evaluation action/constrained flow of actions". For example, if an Action of a Scenario is classified as an "information provision", there must be another Action, that constitutes the corresponding "information request".

For discovering *alternative* or *complementary* Goals, the Parameters of a Goal are inspected. The first rule for alternatives is based on the idea that any Parameter of the given Goal can have an alternative. Then, any non-existing combination of alternative Parameters will provide a candidate for an alternative Goal. The second rule for alternatives attempts to complete missing branches in the path of Actions of a Scenario. If an Action is only executed under a certain condition, there should be another Action, being performed in the case the condition is false. Information to complete the missing branches can be taken from candidate alternative Goals. These Goals share all except one Parameter, and the Verb, but are different in the Manner.

The first rule for composition is based on the idea that if a Scenario is to be repeatable, its should not leave the system in a state so that the Scenario cannot be executed again. The meta-model is vague regarding this rule. The rule states that a Scenario has multiple Initial and Final States, whereas the meta-model provides no cardinality. Furthermore, the explanation of the rule relates the Scenario's Initial and Final States to the involved Agents, a connection that is also not shown in the meta-model. According to this rule, all initial states of a Scenario should also be its final states. For any initial state that is not, the modeler should come up with a recovery Scenario and corresponding Goal. The second rule for composition entails reasoning about the Resources related to Scenario Actions. For every Action that consumes a Resource, there should be an action to produce that Resource. If none such action is present in the `Scenario`, this leads to the identification of new `Goals`, that can be addressed by additional complementary `Scenarios`.

The B-SCP framework (Bleistein et al. 2006) provides an elicitation strategy on organizational IT requirements. First, the business context is modeled with the contextual part of Jackon problem diagrams. Participants of the business process are modeled as domains of interest, their relationships as shared phenomena. From the context model, strategic requirements are modeled with a subset of i* diagrams. Abstract business goals are modeled as i* soft goals, which are broken down into hard goals and then tasks. The goal model and the context model are interconnected with requirement constraints and references in a variant of a Jackons problem diagram. Finally the business process is modeled with role activity diagrams, whose constitutuents refer to the appropriate parts of the goal-context model.

i* (Yu 1997) implicitly provides an elicitation strategy as well. It suggests to determine "intentional actors" first. Then the inter-actor dependencies are modeled in a "strategic dependency model". Dependencies express how actors depend on other actors performing tasks or providing resources in order to achieve goals. Up to that point, actors are considered black boxes. In the next step, a strategic rationale model is created. The modeler looks "inside" the actors to uncover their intentions. Some of the intentional elements of the SD model can be allocated to certain actors. These coarse-grained intentional elements that were expressing inter-actor dependencies are then decomposed to obtain a better understanding of the rationale of certain dependencies.

As mentioned before, the literature from which we extracted the strategies described above, apart from the works on KAOS, does not explicitly refer to them using a standard terminology. Therefore, existing strategies are hard to identify, evaluate, and compare. With a framework to formalize elicitation strategies, strategies could become an explicit part of a language specification. In addition, the intended use of the language would be illustrated in standard terminology. This would make comparison of the strategies easier. It would also contribute to the comparison of the languages, as the meta-model of the language has a strong impact on the number of strategies that are available. We therefore propose the construction of a framework to support model-driven requirements strategies, which is outlined in Chapter 5.

4. Unified Requirements Modeling Language (URML)

This chapter presents the **U**nified **R**equirements **M**odeling **L**anguage. The requirements for URML are discussed in Section 4.1. Section 4.2 is focused on the abstract syntax of URML but also introduces the notation and illustrates the use of the language with excerpts from an example model. At the beginning of the section, the structure of the URML meta-model is described in terms of packages. An overview of the core abstractions of URML is given. Each package is then described in a separate subsection. The partition of the URML abstract syntax that represents a package is again partitioned into multiple meta-model diagrams. The abstractions presented on the meta-model diagrams are then defined and examples on their use in URML diagrams are given. The examples are taken from a model that describes the phlebotomy process in a hospital. The phlebotomy process is a complex system, in which multiple systems and persons participate. It contains workflows dealing with everything from the order of a blood test by a physician to the disposal of the analyzed blood sample. A discussion of metamodeling issues concludes the section.

The concrete syntax of URML is then defined more formally in Section 4.3. The concrete syntax meta-model is presented and the mapping of abstract to concrete syntax is discussed. Then follows a discussion regarding how URML reflects Moody's design principles. The chapter is concluded by comparing URML to URN in Section 4.4 and to KAOS in Section 4.5. An unpartitioned version of the abstract syntax meta-model is provided in Appendix Section A.1, as well as further specification details in Appendix Sections A.4 and A.5.

4.1. Requirements for URML

The main purpose of URML is to support early requirements elicitation. This entails support for scenarios in which analysts elicit requirements knowledge from any form of document provided by stakeholders as well as person-to-person situations in which the results of an interview or a brainstorming session are to be recorded by analysts and reviewed by stakeholders. In both kinds of scenarios, the elicitation shall be guided by the abstract syntax meta-model, via the abstractions contained therein.

The abstractions provided by URML shall cover a broad range of knowledge areas that are relevant to requirements engineering. It is of interest which persons, organizations, both of them in a specific role have a stake in the system, either because they have something to gain or to lose of the system to be constructed, or because they will be the ones using it. For those, it is - in certain limits - of interest what is particularly important to them and could be put at stake by the system, be it property or intellectual goods. Also the intentions of persons regarding the system under constructions influence the requirements to a great deal.

However, it is often difficult to uncover these intentions early on. Intentions may be driven by very individual concerns like preferring one color over the other. Sometimes intentions are political, sometimes driven by business concerns. If intentions are not communicated because they are not known or not disclosed on purpose, URML shall provide other means to express knowledge that drives requirements. If the system to be developed is potentially part of a product line, product variability in terms of features is a great concern. In certain domains safety considerations are of utmost importance, in order to avoid injury or loss of life. In other domains the security of the system under discussion has to be ensured so that no malicious persons gain access to information they are not authorized to see, or functions they are not authorized to trigger. Particularly in such scenarios, but also in more general ones, the flow of goods or of information through the system under discussion may be something stakeholders want to discuss.

The system under discussion never has full control over its environment, therefore it is important how it can react to certain undesired situations, whose occurrence it might be unable to avoid. Sometimes, hazardous situations may be dealt with provisions external to the system under discussion. For example, a hazard may be circumvented by establishing conventions how a system or its parts are to be used. In domains where ease of use has high priority, it is particularly important to arrive at a good understanding how the system is used by its users, and how it helps them to achieve their tasks, with a certain quality. The environment of the system under discussion is often not only its users, but also other systems, so it can be important to get on overview over the interfaces over which the interaction occurs. If a system to be developed will be plugged into an existing larger system or process, it is important to understand the surrounding process, for example a particular a business process of a company ordering the system.

Finally, the requirements analyst needs some degrees of freedom, to record knowledge even if he is unable immediately categorize it into one of the aforementioned areas.

All requirements knowledge expressed with URML shall be traceable to functional and quality requirements. For any requirement it shall be expressible wether it is driven by regulatory concerns. Lightweight traceability shall be offered by enabling links to requirement knowledge expressed in other languages, potentially maintained with other tools than a URML-implementing tool.

In spite of valuing one particular view on requirements knowledge over the other, URML shall provide a holistic approach by integrating abstractions known from other requirements modeling languages. By providing a broader range of abstractions, it shall allow for a greater range of entry points into a discussion about requirements. Instead of being goal-, feature, or use case-driven, it shall facilitate flexible elicitation strategies for the analyst. To avoid analysis paralysis, URML shall allow to record stakeholder requests “as-is”, i.e. as not further classified knowledge. The primary goal here is not to lose any information. The recorded knowledge can be refined into other URML abstractions

later in the elicitation process - not hindering the elicitation workflow is considered more important than correctly classifying the requirements knowledge as early as possible.
The notation of URML shall follow Moody's design principles for visual notations. The ultimate goal of URML is that URML diagrams support the communication of requirements knowledge to stakeholders, even if they are no requirements engineering experts. The approach to achieving easily recognizable visual elements is to use iconographic symbols to represent entity instances.
Finally, URML shall satisfy the following constraints: The specification of URML shall follow the architecture outlined by Selic (Selic 2011) and Kleppe (Kleppe 2008). The abstract-to-concrete syntax mapping shall follow the approach proposed by Fondement and Baar (Fondement and Baar 2005). For pragmatic reasons, the reference implementation of URML shall be done for the Enterprise Architect CASE tool.

4.2. URML Abstract Syntax

The URML comprises a total of six packages: The *Kernel package*, the *Danger package*, the *Feature package*, the *Stakeholder and Goal package*, the *Process and System package*, and the *Requirements package* (see Fig. 4.1).

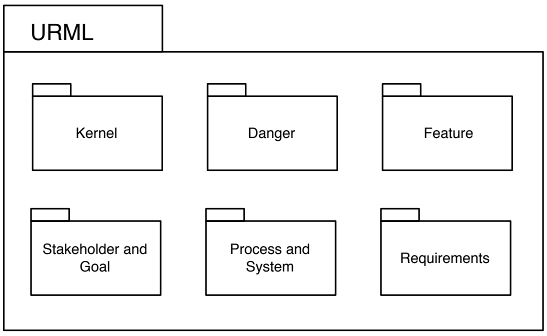


Figure 4.1.: The packages of the URML. Relationships were omitted here.

Each package contains elements needed to model a specific aspect of a URML-based requirements specification. This section gives an overview of the packages. The Kernel package comes first, as it is needed as the syntactical base for all other packages. All other packages are dependent on the Kernel package (see Fig. 4.2).

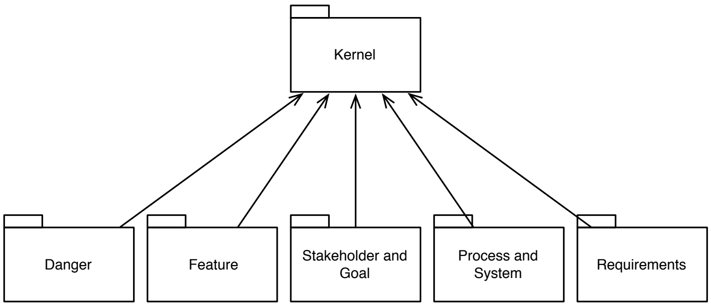


Figure 4.2.: The Kernel package provides the base abstractions for the other packages of URML

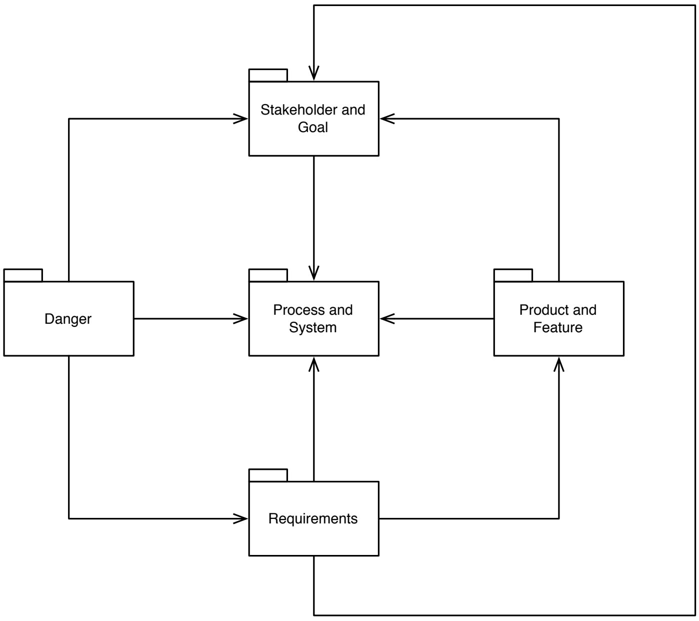


Figure 4.3.: The relationships between the non-Kernel packages of URML

The architecture of URML apart from the Kernel is shown in Fig. 4.3. Every package has three to five dependencies to other packages. The concrete dependencies are not discussed here, as the according abstractions are yet to be discussed in the following subsections. With basic knowledge about URML's abstractions, Figure A.2 can be reviewed.

The packages *"Danger" and "Stakeholder and Goal"* work together to show how danger affects stakeholders. The packages *"Danger" and "Requirement"* cooperate to model the mitigation of dangers through requirements. The packages *"Danger" and "Process and System"* show how processes are vulnerable to and can trigger dangers. The packages *"Process and System"* and *"Requirement"* describe how requirements detail processes or constrain them. The same purpose but on a different level of detail is provided by the *"Requirement" and "Feature"* packages. *The "Requirements" package works together with the "Stakeholder and Goal"* package to address stakeholder requests and goals with requirements. A similar statement can be made by *"Stakeholder and Goal" and "Feature"*, on a different level of detail. *"Stakeholder and Goal" with "Process and System"* relate the concept of an actor with that of a system and thus help with modeling system interaction. *"Feature" and "Process and System"* show how features of a system enable a process and how a feature pinpoints functionality of a system. An alternative model of the structure of the URML meta-model is shown in Figure 4.4.

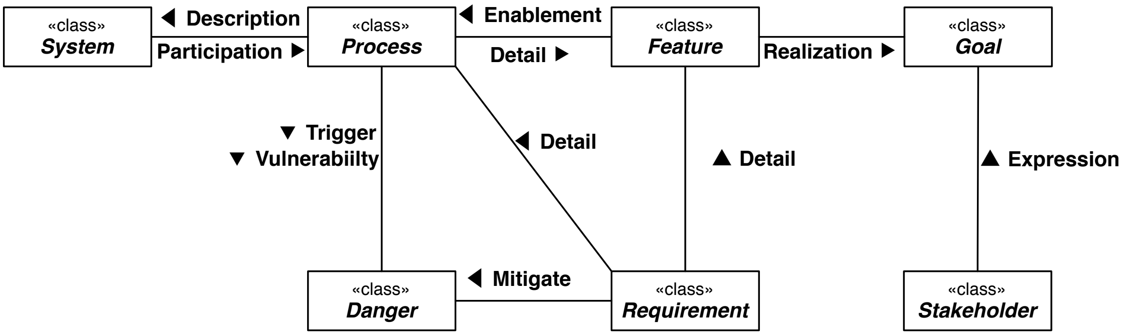


Figure 4.4.: The main concepts of the URML in a simplified model

In the following subsections we describe the contents of each package in more detail. Some meta-classes appear in multiple package descriptions, but not necessarily with all their attributes. In the description of a package, only the attributes and relationships of relevance to the package are explained.

4.2.1. Kernel

The URMLModel meta-class groups together all elements of a model[1]. It is the starting point for every URML model. It has a name and a description. The base class for all elements of a model is URMLModelElement. All elements of a URML model have in common that they can have a plain text description. The abstract URMLModelElement is specialized to two more concrete classes, the URMLModelRelationship and the URMLModelEntity. URMLModelRelationship is the superclass of all relationships of the meta-model, and URMLModelEntity is the superclass of all entities of the URML. URMLModelEntity has a name attribute. Each subclass of URMLModelEntity participates in one or more subclasses of URMLModelRelationship. Every concrete subclass of URMLModelEntity can participate in two relationships, Inheritance and Aggregation. More specific relationships are introduced in the other packages.

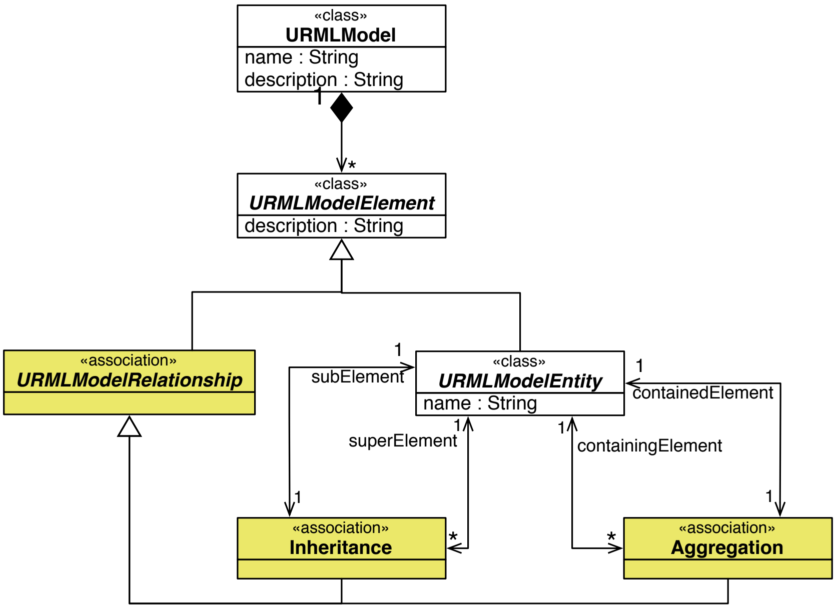
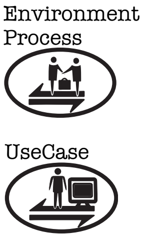


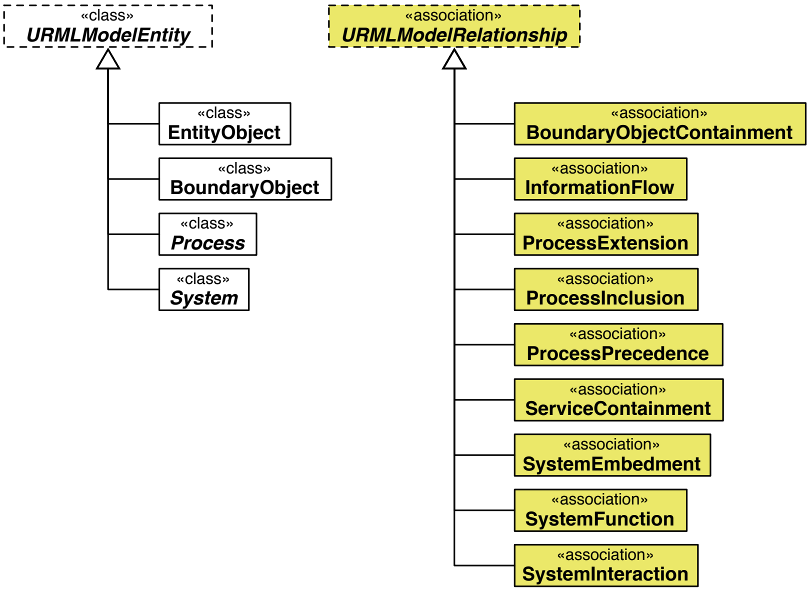
Figure 4.5.: The URML Kernel package

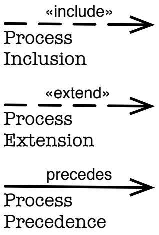
[1] like the “unique starting symbol” in the definition of a formal grammar

4.2.2. Process and System



The Process and System package introduces four meta-classes that inherit from URMLModelEntity and nine meta-classes that inherit form URMLModelRelationship (Fig. 4.6). EntityObject, BoundaryObject, Process and System inherit from URMLModelEntity. Process and System are further specialized within the package. BoundaryObjectContainment, InformationFlow, ProcessExtension, ProcessInclusion, ProcessPrecedence, ServiceContainment, SystemEmbedment, SystemFunction, and SystemInteraction inherit from URMLModelRelationship.





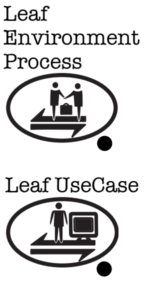
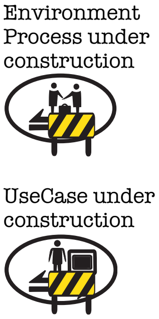


Figure 4.6.: Subclasses of URMLModelEntity and URMLModelRelationship, introduced in the Process and System package.


Environment Process under construction
UseCase under construction

The central concept of the Process and System package is the Process (See Figure 4.7), which models a function of a system. Indirectly, a process models system behavior through its sub-processes. Process is abstract, which means it can not appear on any diagram. Its concrete subclasses are UseCase and EnvironmentProcess. A UseCase is a function of the system under discussion. The set of all UseCases defines the behavior of the system. An EnvironmentProcess is a function external to the system under discussion. It models the environment of a system in terms of external processes in which the system under discussion is embedded. UseCase and EnvironmentProcess have much in common: A process can be decomposed through ProcessInclusion relationships. Exceptional or optional extensions of a process are modeled through ProcessExtension relationships. The order in which processes can occur (sometimes called control flow, or workflow) is modeled by ProcessPrecedence relationships, which realize a partial ordering of processes. As proposed by Jacobson and implemented in UML, use cases can't be put into any order, their purpose is to provide an abstract representation of the main usage scenarios of a system. In UML, details about the workflow would be modeled with interaction or activity diagrams. URML as a modeling language for early requirements engineering needed something more lightweight, to allow specification of workflows. The `ProcessPrecedence` relationship enables this on a high abstraction level for URML. The details of the process that are part of system design however can't be modeled in URML, for example how concurrent processes are synchronized, or how exactly a branching condition looks like.

The boolean attribute `leaf` indicates whether - in the given model - a process will be further decomposed or not. This does not necessarily mean atomicity - it is a decision of the modeler when to stop detailing processes. A constraint on `ProcessInclusion` ensures that `Process` instances with `leaf` equal to `true` can't include other `Process` instances. A process has a `precondition` and a `postcondition`, which both are described in natural language. Any process can be marked with a boolean attribute `businessProcess` if it is or is part of a business process. The latter two attributes don't show up on diagrams. A process has an `underConstruction` attribute, which expresses whether the modeler is still working on it.

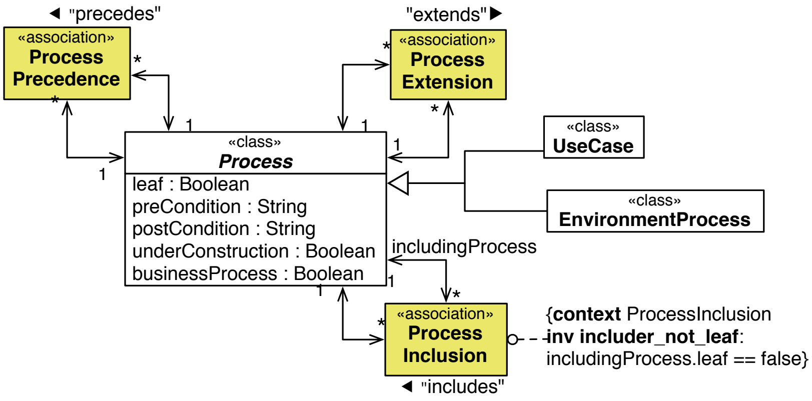


Figure 4.7.: A `Process`, its subclasses, and its relationships.

The figure taken from the case study (see Figure 4.8) shows four instances of `UseCase`. The `UseCase` "Blood Sample Management Process" is related to each of the lower `UseCase` instances ("Pre-Analytical Phase", "Analytical Phase", and "Post-Analytical Phase") with an instance of `ProcessInclusion`. This means that the blood sample management process of a typical hospital can be divided into the pre-analytical phase, the analytical phase, and the post-analytical phase. Of these three lower `UseCase` instances, each two adjacent pairs are related by an instance of `ProcessPrecedence`. This means that the "Pre-Analytical Phase" `UseCase` precedes the "Analytical Phase" `UseCase`, which in turn precedes the "Post-Analytical" `UseCase`. Figure 4.9 shows how the diagram of Figure 4.8 is an instance of the meta-model in Figure 4.7.

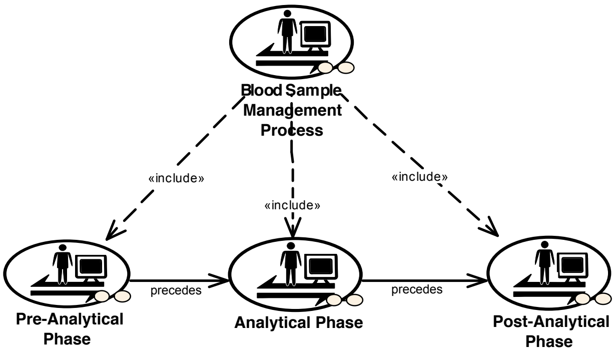


Figure 4.8.: The high-level workflow of the blood sample management process

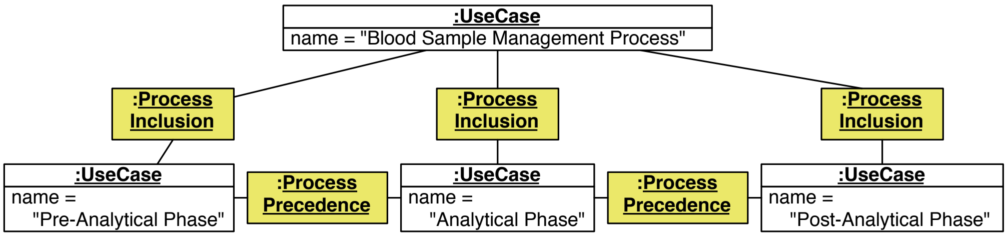

:UseCase
name = "Blood Sample Management Process"
:Process Inclusion
:Process Inclusion
:Process Inclusion
:UseCase
name = "Pre-Analytical Phase"
:Process Precedence
:UseCase
name = "Analytical Phase"
:Process Precedence
:UseCase
name = "Post-Analytical Phase"

Figure 4.9.: A UML instance diagram corresponding of the previous figure.

The next diagram from the running example contains four instances of `UseCase`. The "Sample processing" `UseCase` is related to the bottom right `UseCases` with an instance of `ProcessExtension`. It is related to the top right `UseCase` by an instance of `ProcessInclusion`. Figure 4.10 shows the "Sample Processing" use case, which is by the way part of the analytical phase (not shown on diagram). From the perspective of the laboratory, sample processing always includes monitoring the blood analyzer device. If the blood analyzer has errors, or if a sample is broken, the operator needs to react accordingly (e.g. ordering a new sample, calling maintenance technician). During sample process, things can go wrong that are external to the analyzer device, which takes different error handling procedures (e.g. power outage).

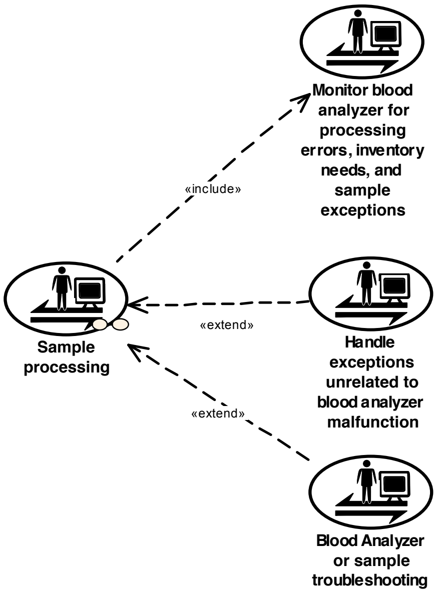

Monitor blood analyzer for processing errors, inventory needs, and sample exceptions
«include»
Sample processing
«extend»
Handle exceptions unrelated to blood analyzer malfunction
«extend»
Blood Analyzer or sample troubleshooting

Figure 4.10.: Exceptional and standard behavior of the "Sample Processing" `UseCase`

The next excerpt of the running example (Figure 4.11) shows five instances of `EnvironmentProcess`. The lower four instances are `leaves`. This gives an example where the modeler decided not to go into further detail one level below the "Prepare Sample Draw" `UseCase`. That means that e.g. the `UseCase` "Select venipuncture site" will not be further detailed in the model in terms of inclusion or extension relationships.

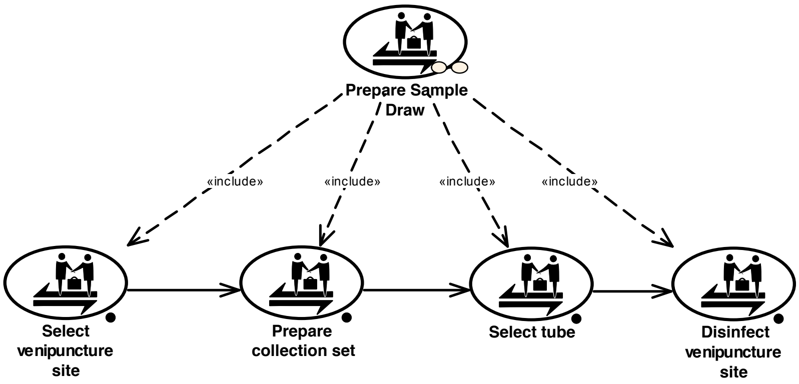


Figure 4.11.: Leaf use cases

The URML supports three different types of `Systems`. The `SystemUnderDiscussion` class reflects the system for which the requirements are to be engineered. The `Actor` represents a role that a person or a system external to the system under discussion do play. Actors are explained later in this section. Internal to the system under discussion, other systems, whose internals are not known to the modeler, are modeled by ServiceProvider. Service providers are part of the system under discussion. The construct can be used to model internal components whose intrinsics cannot be modeled, either because they are human, or because they are commercial off-the-shelf (COTS) components. A ServiceProvider can be further characterized via the type attribute that relates to the ServiceProviderType enumeration. If the type is Human, it models humans that are working in the system under discussion. If its is Software, it models software being used therein, and if it is System, it models a complex system, possibly (but not limited to) a mechanical systems.

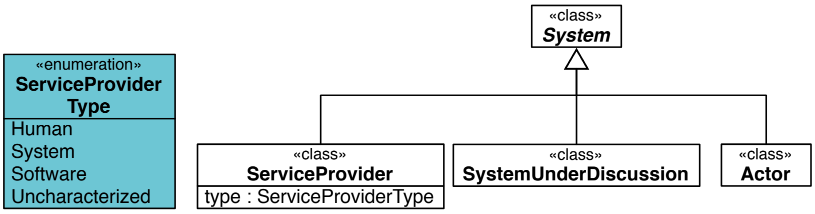


Figure 4.12.: The three system types and further sub-categorizations of ServiceProvider

That a SystemUnderDiscussion uses a ServiceProvider can be shown through the ServiceContainment relationship.

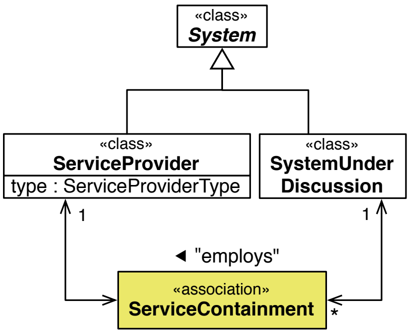
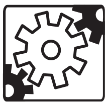


Figure 4.13.: A SystemUnderDiscussion can participate in many ServiceContainment relationships, each relating a SystemUnderDiscussion to a ServiceProvider

In Figure 4.14 of the running example, the clinical chemistry laboratory is an instance of SystemUnderDiscussion. It is related to three instances of ServiceProvider, the “Hospital Information System”, the “Laboratory Information System”, and the “Blood Analyzer”. The first two service providers are software systems. So the type attribute of these service providers is set to Software. The blood analyzer is a system consisting of mechanical, electrical, and software components, thus its type is set to System. The instances of ServiceProvider are related to the SystemUnderDiscussion via ServiceContainment relationships. This means that it uses the services of three service providers: The hospital information system is used by physicians to order tests.

System Under Discussion



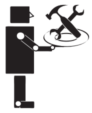
Actor

Service Provider (Human)

Service Provider

Service Provider (Software)

Service Provider (System)



uses →
Service Containment

These test orders are usually transmitted to the laboratory information system. The laboratory information system collects the results of tests and reports them back to the hospital information system. The tests are nowadays done by blood analyzer systems. A blood analyzer, as opposed to the other two service providers, not only consists of software but complex mechanical and electrical components. This difference is underlined by the different icon, which reflects that the value of the type attribute is `System` for the Blood Analyzer.

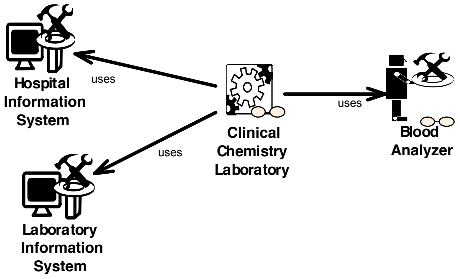

Hospital Information System
uses
Clinical Chemistry Laboratory
uses
Blood Analyzer
uses
Laboratory Information System

Figure 4.14.: The `SystemUnderDiscussion`, the Clinical Chemistry Laboratory, uses three `ServiceProviders`, two of type `Software` (Hospital and Laboratory Information System) and one of type `System` (Blood Analyzer)

After `SystemUnderDiscussion` and `ServiceProvider`, the third type of system is the `Actor`. `Actor` instances model systems that are external to the system under discussion. An actor is a role of something that actually uses (or interacts with) the system under discussion. Very often actors are roles that a person can assume. Actors can however also be roles of other technical systems. Actors show who or what will make use of the system under discussion. To model the interaction of an actor with a system, we have to introduce the term `BoundaryObject` first.

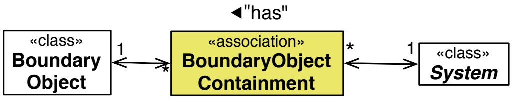


Figure 4.15.: The interface of a `System` is the set of `BoundaryObjects` it contains. The containment is modeled with `BoundaryObjectContainment` relationships.

A system can contain various instances of `BoundaryObject`, which enables modeling the interaction of external systems (like actors) with the system. A BoundaryObject represents a part of the interface of a system. Interaction with systems can only occur through instances of BoundaryObject.

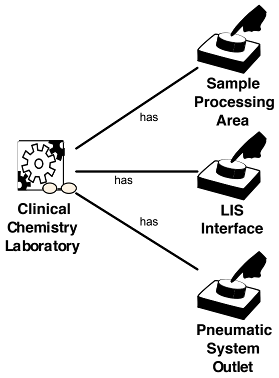
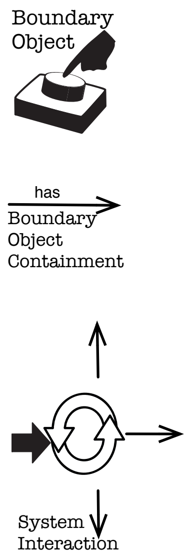


Figure 4.16.: One instance of `SystemUnderDiscussion` with three instances of `BoundaryObject`. The instances of `BoundaryObject` are related to the `SystemUnderDiscussion` via `BoundaryObjectContainment` relationships.



In Figure 4.16, some boundary objects of the chemistry laboratory are shown. Samples that are transported to the lab are collected in the sample processing area. If the hospital has a pneumatic tube system for rapid transport of samples from e.g. the emergency room, there must be an outlet where tubes loaded with blood samples arrive. The laboratory information system (LIS) should have an interface with the hospital information system so that the two can communicate.

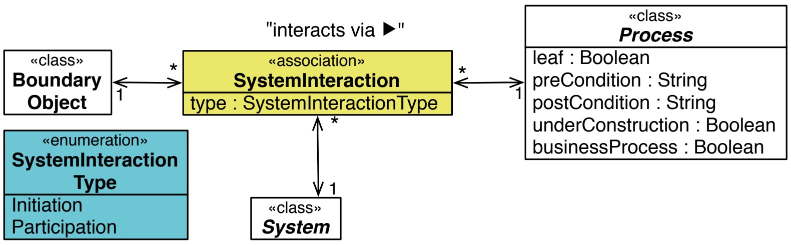


Figure 4.17.: The nary SystemInteraction relationship relates `Systems` with `Processes` and `BoundaryObjects`. Whether a `System` participates in the `Process` or initiates it, can be recorded with the `SystemInteraction's type` attribute.

The interaction of one system with another system is modeled by the `SystemInteraction` relationship that relates boundary objects, processes, and systems. So on a URML diagram, the systems that interact are not connected directly, but indirectly through this relationship that shows how one system uses a boundary object to participate in or initiate (`SystemInteractionType`) a process of the other system.

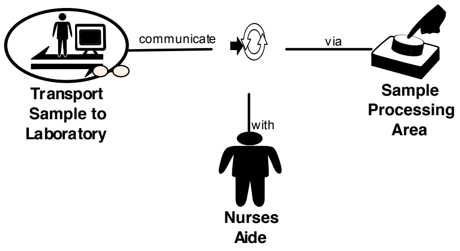


Figure 4.18.: One instance of UseCase , one instance of Actor, and one instance of BoundaryObject. The icon connecting the three is an instance of the SystemInteraction relationship.

Figure 4.18 shows how a nurses aide interacts with the laboratory by transporting blood samples to the sample processing area of the lab. The difference between UseCases and EnvironmentProcesses is reflected in the SystemFunction and SystemEmbedment relationships. The system under discussion is to be used in environment processes. This can be shown via the SystemEmbedment relationship. In order to relate the system under discussion with its functions, the SystemFunction relationship can be used.

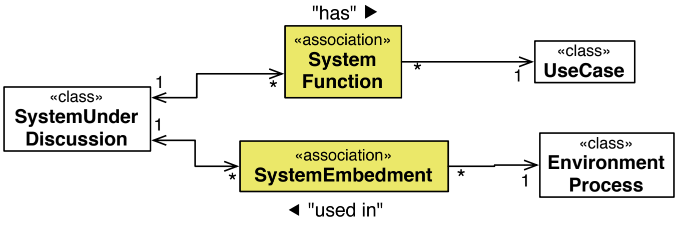

"has" ▶
«association» System Function
«class» UseCase
«class» SystemUnder Discussion
«association» SystemEmbedment
«class» Environment Process
◀ "used in"
1
*
*
1
1
*
*
1

Figure 4.19.: The SystemFunction and SystemEmbedment relationships, relating the SystemUnderDiscussion with UseCases and EnvironmentProcesses

The system under discussion has two more important relationships with abstractions describing kinds of processes (more on this in the following section): A system is usually embedded in a certain environment, which also has processes. This is modeled with the SystemEmbedment relationship between SystemUnderDiscussion and EnvironmentProcess. Conversely, a system contains functions by itself. This is modeled with the SystemFunction relationship between SystemUnderDiscussion and UseCase. How the two process subclasses relate to each other will be shown in the next section.

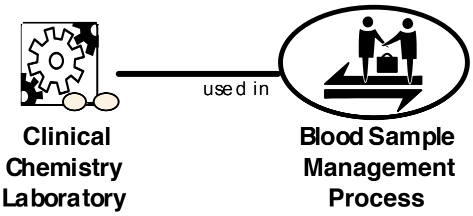
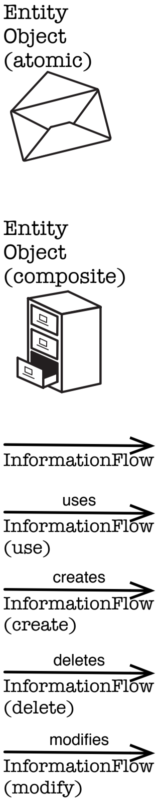

used in
Clinical Chemistry Laboratory
Blood Sample Management Process

Figure 4.20.: One instance of `SystemUnderDiscussion` and one instance of `Environment-Process`, related by an instance of the `SystemEmbedment` relationship

Figure 4.20 shows that the clinical chemistry laboratory is embedded into the blood sample management process (of the hospital). Apart from the analysis of samples, another function of a lab could be the training of new staff (See Figure 4.21).



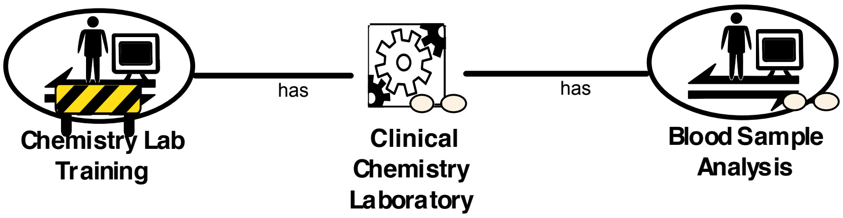


Figure 4.21.: One instance of `SystemUnderDiscussion` and two instances of `UseCase`, with each `UseCase` related by an instance of the `SystemFunction` relationship. For the left `UseCase` instance, the value of `underConstruction` is set to true.

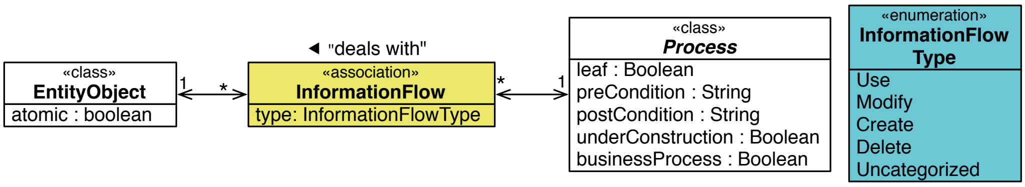


Figure 4.22.: `InformationFlow` to, from, or between `Processes` means that `EntityObjects` are being used, modified, created, or deleted.

Data flow can be modeled through `InformationFlow` relationships that connect a process to a set of `EntityObject` instances. How a process actually affects an entity object used in an information flow relationship can be characterized by the `InformationFlowType` enumeration. To mark objects that are not decomposable anymore, the EntityObject has an attribute called atomic. This allows to have different icons for atomic and composite EntityObjects. The attribute leads to a required constraint on a model using it: Atomic EntityObjects may not aggregate other EntityObjects then.

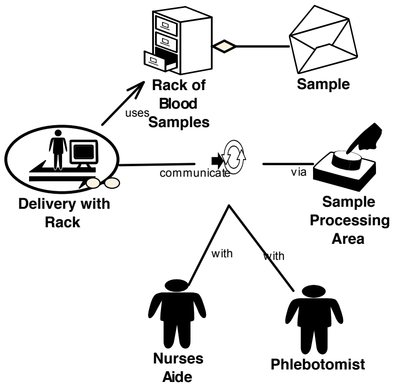


Figure 4.23.: Two instances of `Actor`, one instance of `UseCase`, and one instance of `BoundaryObject` related by one instance of one instance of `SystemInteraction`. The `composite EntityObject` is related to the `UseCase` via an `InformationFlow` instance. The `atomic EntityObject` is related to the `composite EntityObject` via an `Aggregation` instance.

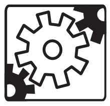
Figure 4.23 shows how blood samples can be delivered to the chemistry laboratory. A nurses aide or a phlebotomist can use a rack to transport the samples to the laboratory, where they are received and processed in the sample processing area.

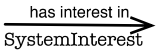
4.2.3. Stakeholder and Goal

The `Stakeholder and Goal` package introduces six meta-classes that inherit from `URMLModelEntity` and eleven meta-classes that inherit form `URMLModelRelationship` (Fig. 4.24). `AssessmentSketch`, `Asset`, `Goal`, `Idea`, `Request` and `Stakeholder` inherit from `URMLModelEntity`. `Goal` is further specialized within the package. `AssetOwnership`, `GoalContribution`, `GoalDecomposition`, `GoalRealization`, `GoalStatement`, `Mention`, `Motivation`, `RequestCause`, `RequestStatement`, `StakeholderHierarchy`, and `TestDescription` inherit from `URMLModelRelationship`.

Stakeholder (Uncategorized)

System Under Discussion





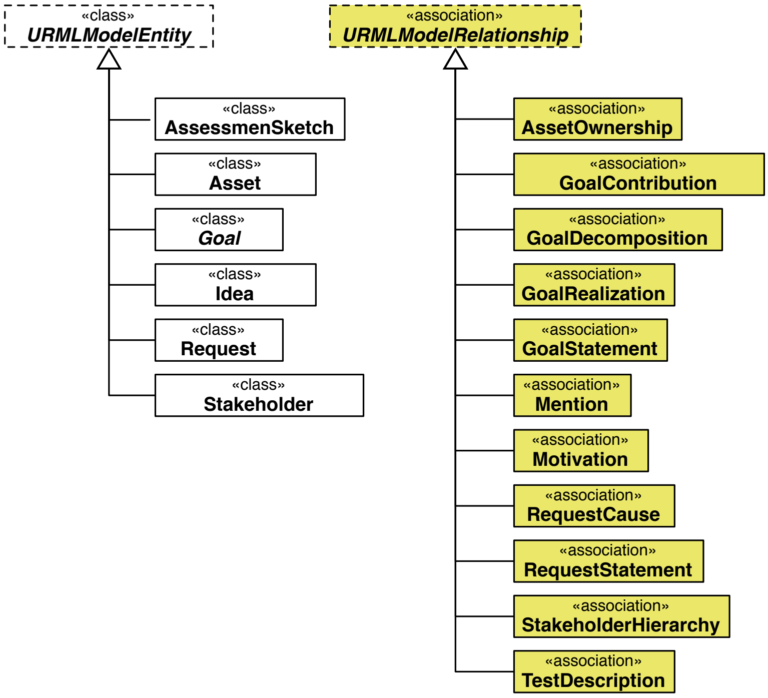


Figure 4.24.: Subclasses of `URMLModelEntity` and `URMLModelRelationship`, introduced in the `Stakeholder and Goal` package.

The `Stakeholder` concept is used to model the interest (or the stake) of somebody or a group of persons in the system under discussion (`SystemInterest`). Stakeholder models a role of that person or group of persons. It models not only the who, but also the why of the interest in the system. Quite often however, a stakeholder can be mapped to a single person. During requirements elicitation, stakeholders are interviewed in order to determine their “stake”, then the requirements analyst can identify which kind of stakeholder he is talking to. To facilitate tracking to which concrete persons the requirements analyst talked, the `interviewedPersons` attribute can be used. `Stakeholders` can be given a `weight` value, which can be used to rank stakeholders.

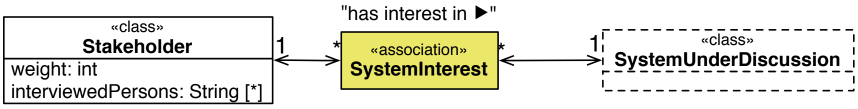


Figure 4.25.: `Stakeholders` are interested in the `SystemUnderDiscussion`

Figure 4.26 shows five stakeholders of the clinical chemistry laboratory: The phlebotomist wants to be sure that his work (drawing samples) is properly continued in the lab (analyzing samples). The physician orders a test in order to help with the diagnosis of the patient. The patient hopes on reliable results of the laboratory. The Nurse sometimes has to transport samples to the laboratory. The example already shows two different types of stakeholder.

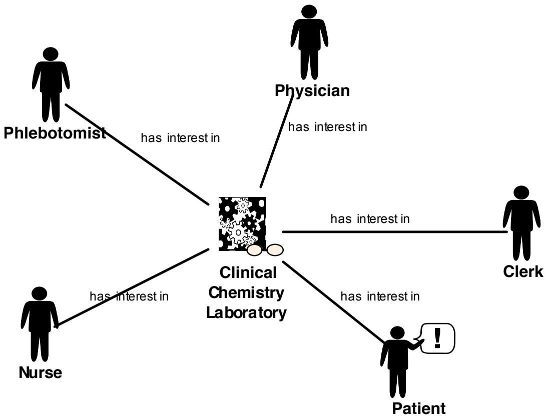


Figure 4.26.: One instance of `SystemUnderDiscussion` with five instances of `Stakeholder`, related by instances of the `SystemInterest` relationship. Actually four of the stakeholders are instances of subclasses of stakeholder, which will be explained in the following paragraph.

The URML offers four classes of `Stakeholder`. An `Actor` is something or somebody interacting with (i.e. using) the system. Actors typically have objectives like “the system should enable me to efficiently perform my daily tasks”. A `BusinessStakeholder` is somebody ordering or procuring the system. Especially regarding large business systems, this role is often different from the users (i.e. Actors) of the system: The person ordering the system will not use it, but orders the system on behalf of a group of users, possibly because that stakeholder is a manager to which these users are reporting to. In large organizations, there may be a third role: The `Customer`, is the role of somebody buying the system. The person in charge of paying for a system usually has very different objectives than the other roles. While the previous roles have interest in a large feature set, the `Customer` rather is interested in cost effectiveness. If none of the aforementioned classes is found to be suitable, the `Stakeholder` class can be used to represent a generic type of stakeholder. A (subjective) assessment of the stakeholder's importance to the project can be modeled through the `weight` attribute. The relationships between different stakeholders can be modeled with the `StakeholderHierarchy` relationship.

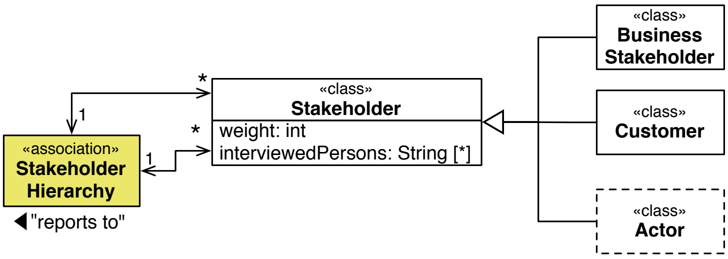


Figure 4.27.: `Stakeholder`, its three subclasses, and the stakeholder-specific relationship `StakeholderHierarchy`.

Figure 4.28 shows that a nurse has more responsibilities than a licensed practical nurse (LPN), which has more responsibilities than a nurses aide. That is why the stakeholder hierarchy goes into the opposite direction: Nurses aides report to LPNs, which in turn report to nurses. With the URML , statements uttered by stakeholders can be classified into three types.

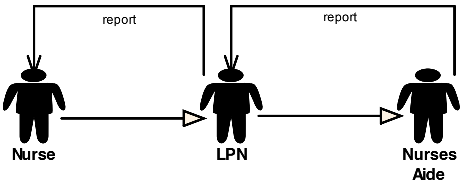
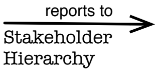
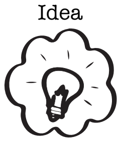
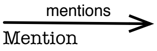


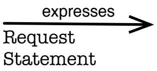
Figure 4.28.: Three instances of `Actor`, connected by instances of `Inheritance` and `StakeholderHierarchy` relationships.

A `Request` is a Stakeholder statement that the requirements engineer has to translate into requirements[2] Which stakeholder made which request can be shown with the `RequestStatement` relationship. `Ideas` model general knowledge about the world that the stakeholder may `Mention`. Ideas can be annotated by a URI that points to a source of that idea (e.g. a web page, or a book). `Ideas` can become the `Motivation` for stating `Requests`. Ideas should be modeled in order to help with the understanding of requests. Knowledge about the world can make a request seem reasonable, but there is a specific kind of knowledge about the world that the URML highlights: the objectives of the stakeholders.









[2] See section 4.2.6.

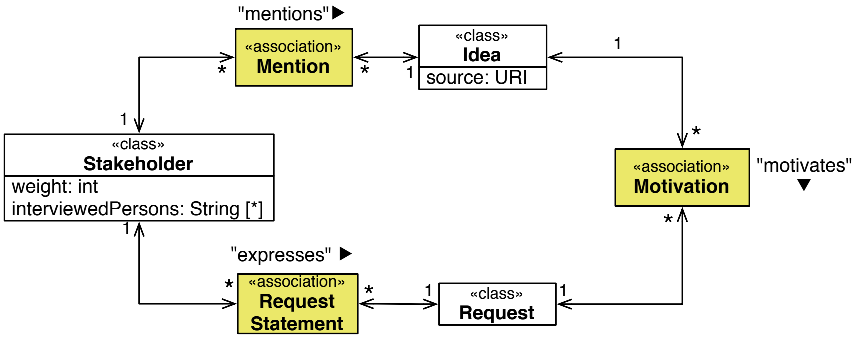


Figure 4.29.: A `Stakeholder` mentions `Ideas` and states `Requests`. Certain `Ideas` can motivate `Requests`.

In the URML, these are called `Goals`, which also can be seen as the cause for requests (`RequestCause`).

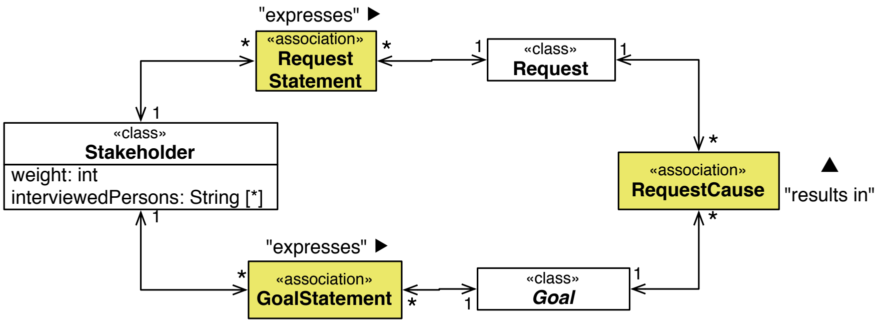


Figure 4.30.: A `Stakeholder` has `Goals`. These can be the cause for `Requests`.

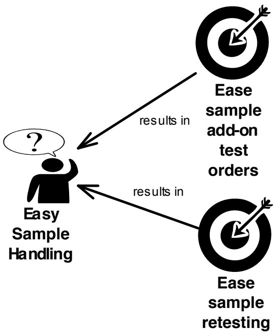
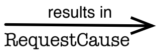


Figure 4.31.: One instance of `Request`, related to two instances of `Goal` via instances of `RequestCause`.

Figure 4.31 shows a request that would affect the post-analytical phase of the blood sample management process. A stakeholder wants to simplify sample retesting. At the same time, he wants to simplify the way add-on tests are done in the system. When talking to the requirements analyst, he might not state the goals in the first place but rather state that the sample handling should not be too complicated. As that is rather vague, the analyst could ask the stakeholder then which goals are behind that request. Goals stated by the stakeholder (`GoalStatement`) can be characterized into two types: hard goals (`Goal`) and soft goals (`SoftGoal`).



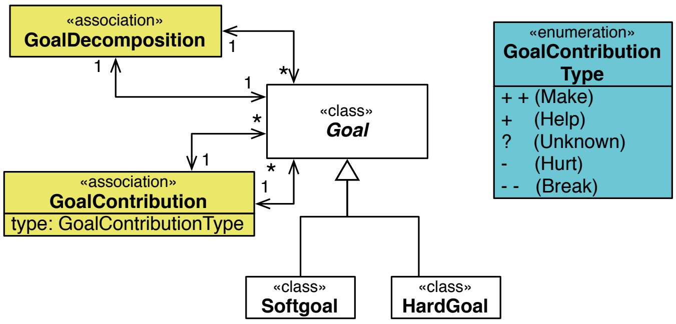
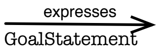
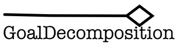
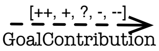


Figure 4.32.: Hard and soft goals. Goals can be decomposed. Apart from that the influence each other via `GoalContribution` relationships.

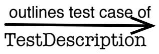


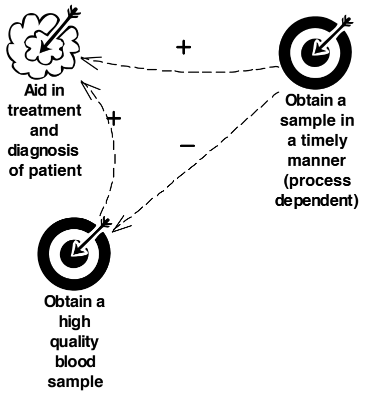




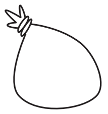
For modeling complex goals, goals can be decomposed (`GoalDecomposition`). During the analysis of goals, the modeler will very likely detect that goals often interfere with each other. To keep track of that, he can model how one goal contributes to the other with a `GoalContribution` relationship. The effect of the contribution can be positive or negative (`GoalContributionType`). The difference between hard and soft goals is as follows: Soft goals are fuzzy in nature and thus it is impossible to verify them. For hard goals, there must be a clear indicator that a test can be constructed. This is captured by so called assessment sketches (`AssessmentSketch`), that are related only to hard goals (`TestDescription`).

Assessment
Sketch
0.0




+
Aid in treatment and diagnosis of patient
+
−
Obtain a sample in a timely manner (process dependent)
Obtain a high quality blood sample

Asset (Uncategorized)

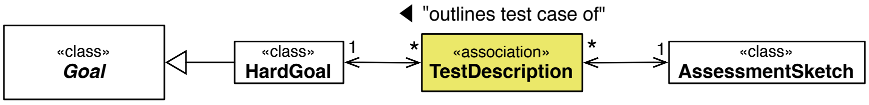


Asset (Financial)

Figure 4.33.: One instance of `SoftGoal` related to two instances of `Goal` via `GoalContribution` relationships with a positive influence (`Help`). The two `Goals` are related to each other by an instance of `GoalContribution` with negative influence (`Hurt`).

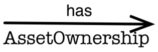
Asset (Identity)

Figure 4.33 shows a soft goal that a physician or a hospital executive might have: blood tests performed in the clinical chemistry laboratory shall aid in the treatment and diagnosis of patients. The goal to obtain high quality blood samples positively contributes to that goal. Collecting samples in a timely manner also contributes to that goal, but there is one caveat: Obtaining samples too quickly may have a negative impact on the quality of the sample.



Asset (Property)

Figure 4.34.: Hard goals must be tested after development. An AssessmentSketch should help with outlining how the test might be done.



`Stakeholders` have `Assets`, which sometimes play an important role in what objectives the stakeholders have, due to the fear of potential losses. More on this topic can be found in the danger package.

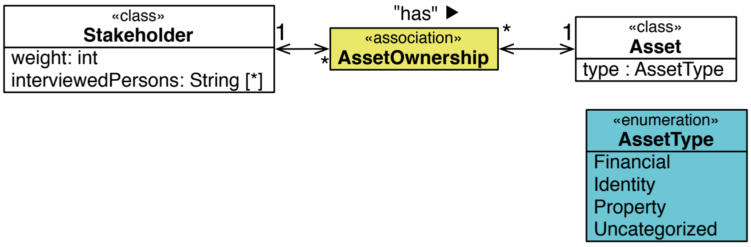


Figure 4.35.: A `Stakeholder` has `Assets`. `Assets` have a `type` attribute whose values are enumerated by the `AssetType` enumeration.

4.2.4. Danger

The Danger package introduces three meta-classes that inherit from URMLModelEntity and four meta-classes that inherit form URMLModelRelationship (Fig. 4.36). Procedure, Danger, and HarmedElement inherit from URMLModelEntity. Danger is further specialized within the package. DangerTrigger, Harm, Mitigation, and Vulnerability inherit from URMLModelRelationship. Mitigation is further specialized within the package.

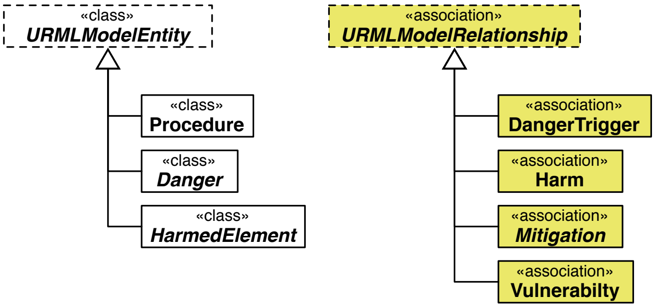


Figure 4.36.: Subclasses of URMLModelEntity and URMLModelRelationship, introduced in the Danger package.

A Danger helps the modeler with expressing that some undesirable can happen. A danger can occur with a certain probability, and when it occurs, a certain severity. There are two subtypes of Danger, that let the modeler differentiate between safety and security issues. The subtype representing safety issues is called Hazard. It has a type, which is by by default set to Uncategorized, but can be set to a different value to differentiate between biological, electrical, chemical, mechanical, meteorological, radiological, seismic, or social types of hazards (HazardType). The danger subtype representing security issues is called Threat.

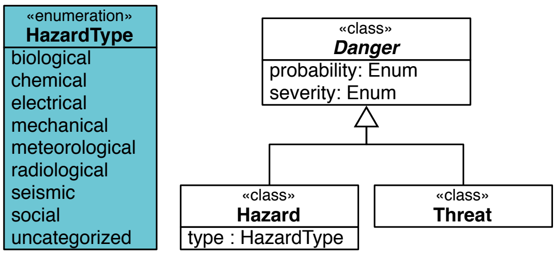


Figure 4.37.: Types of Dangers: Hazard and Threat

A danger potentially affects instances of Asset, Stakeholder, and ServiceProvider. This commonality is expressed by the abstract class HarmedElement. It enables modeling direct and indirect harm to stakeholders. As a stakeholder can be related to various assets

Hazard
(Uncategorized)

Hazard
(Biological)

Hazard
(Chemical)

Hazard
(Electrical)

Hazard
(Mechanical)

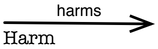
Hazard
(Meteorological)

Hazard
(Radiological)

Hazard (Seismic)

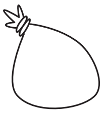
Hazard (Social)

Threat



Harm

Asset (Uncategorized)



Stakeholder (Uncategorized)

Service Provider (Uncategorized)

through the AssetOwnership relationship, it can suffer from danger because one of his or her assets are harmed. In the case of a harmed service provider, the stakeholder is affected because a harmed service provider leads to a dysfunctional system .

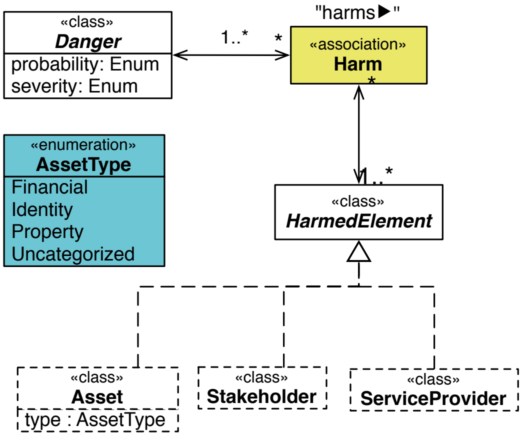


Figure 4.38.: A Danger Harms HarmedElements.

Figure 4.39 shows how a phlebotomist can harm herself. Though the wording of the hazard gives a hint why this can happen, the example is still vague regarding the source of danger.

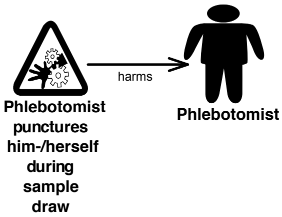


Figure 4.39.: One instance of Actor related to a mechanical Hazard by one instance of Harm.

The source of danger can also be modeled. A Danger can be triggered by a Process, which can be expressed by the DangerTrigger relationship.

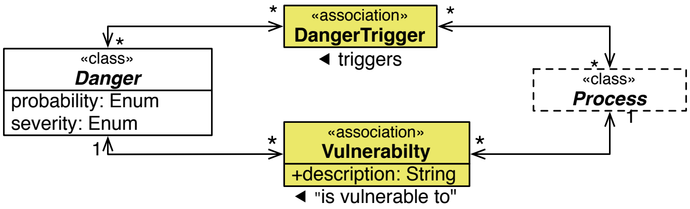


Figure 4.40.: A Process can be vulnerable to Danger, but it can also be the cause of Danger.

Figure 4.41 presents how a process of a blood analyzer system can actively trigger a hazard. Because it needs to aspirate the blood sample in order to create a reagent, it might cross-contaminate samples during that action.

When a danger is not actively triggered, but one ore more system components offer a certain weakness to the external world, this is modeled through a `Vulnerability` relationship. Regarding vulnerabilities it has to be said that the system under construction might not have materialized yet, so modeling vulnerability might as well express a vague feeling as a known weakness of an existing system.

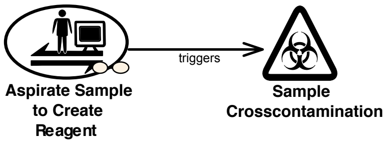
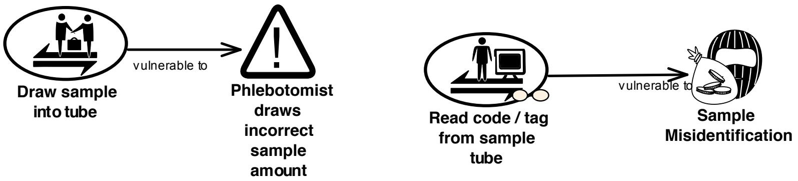


Figure 4.41.: One instance of `UseCase` related to a `biological Hazard` by one instance of `DangerTrigger`.

The two subfigures of Figure 4.42 show how processes can be vulnerable to danger. The first example tells that the phlebotomist might draw the incorrect amount of blood. The second example is about a possible error of a blood analyzer system, that incorrectly maps a code on a sample tube and thus misidentifies to which patient the sample belongs.


vulnerable to
Draw sample into tube
Phlebotomist draws incorrect sample amount
vulnerable to
Read code / tag from sample tube
Sample Misidentification

(a) One instance of `EnvironmentProcess` related to a `uncategorized Hazard` by one instance of `Vulnerability`.

(b) One instance of `UseCase` related to a `Threat` by one instance of `Vulnerability`.

Figure 4.42.: Two examples of `Vulnerability`.

Figure 4.43 shows how understandability of a danger rises when more context is added (compare Figure 4.45). Now it can be seen that the phlebotomist is in danger when withdrawing the needle.

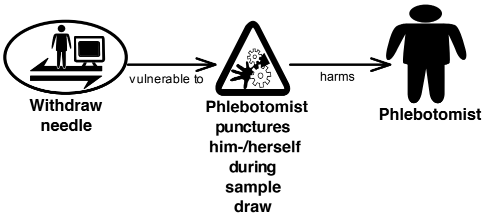
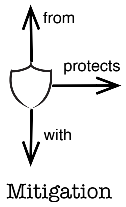


Figure 4.43.: One instance of `Actor` related to a `mechanical Hazard` by one instance of `Harm`. The instances of `UseCase` and `DangerTrigger` show additional context.



Finally, countermeasures for known dangers need to be modeled[3]. There are two possibilities to deal with dangers. The modeler can express that the system under discussion should deal appropriately (e.g. avoid the danger completely, or have contingency plans) with the danger. This is expressed through `RequirementMitigation` relationships that connect a set of dangers, harmed elements, and requirements. Requirements might have been imposed by a regulatory body, exactly to avoid dangers (Not only for this reason requirements may be marked being of `regulatory` nature). It is important to understand that regulatory mitigating requirements are of special importance. Not satisfying these requirements can lead to great jurisdictional issues and heavy fines. Apart from requirements, there may be countermeasures that are not part of the system. These external countermeasures are called `Procedures`. The mitigation of a danger with a procedure is shown then by a `ProceduralMitigation` relationship that connects procedures, dangers, and harmed elements. Both mitigations have in common that they protect harmed elements from dangers. This communality is reflected through the abstract `Mitigation` relationship.

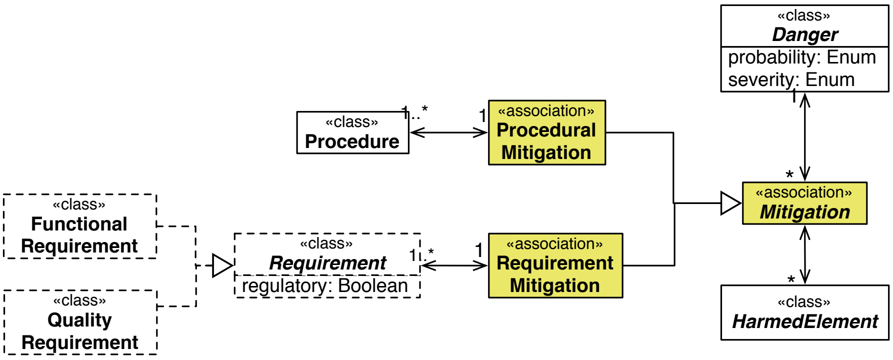


Figure 4.44.: Two kinds of mitigation: `RequirementMitigation` and `ProceduralMitigation`

[3] It is important to mention here that it is impossible to model countermeasures for unknown dangers.

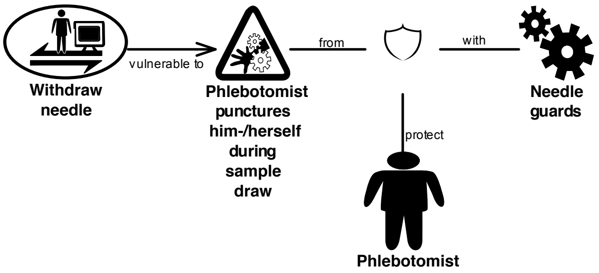


Figure 4.45.: One instance of `Actor` related to a `mechanical Hazard` and a `Functional-Requirement` by one instance of `RequirementMitigation`. The instances of `UseCase` and `Vulnerability` show additional context.

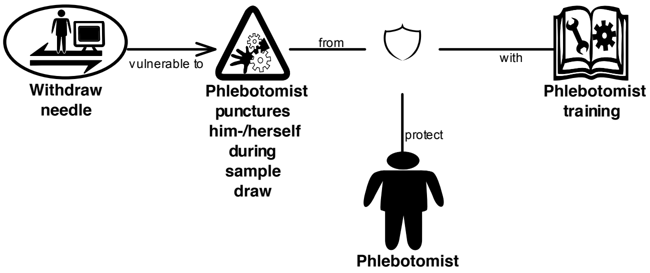
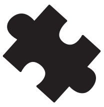


Figure 4.46.: One instance of `Actor` related to a `mechanical Hazard` and a `Procedure` by one instance of `ProceduralMitigation`. The instances of `UseCase` and `Vulnerability` show additional context.

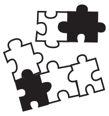
4.2.5. Feature

Feature



Feature (Core)

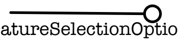
FeatureGroup



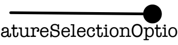
FeatureGroup (Under Construction)

FeatureGroup (Root)

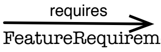
FeatureGroup (Root, Under Construction)



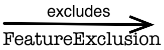
FeatureSelectionOption



FeatureSelectionOption (mandatory)



FeatureRequirement



FeatureExclusion

The Feature package introduces four meta-classes that inherit from URMLModelEntity and ten meta-classes that inherit form URMLModelRelationship (Fig. 4.47). AbstractFeature, Product, ProductLine, and ProductSuite inherit from URMLModelEntity. AbstractFeature is further specialized within the package. FeatureExclusion, FeatureDescriptionUseCase, FeatureList, FeatureRequirement, FeatureSelectionOption, FeatureTree, ProcessEnabling, ProductLineAggregation, ProductLineContainment, and ProductSuiteContainment inherit from URMLModelRelationship.

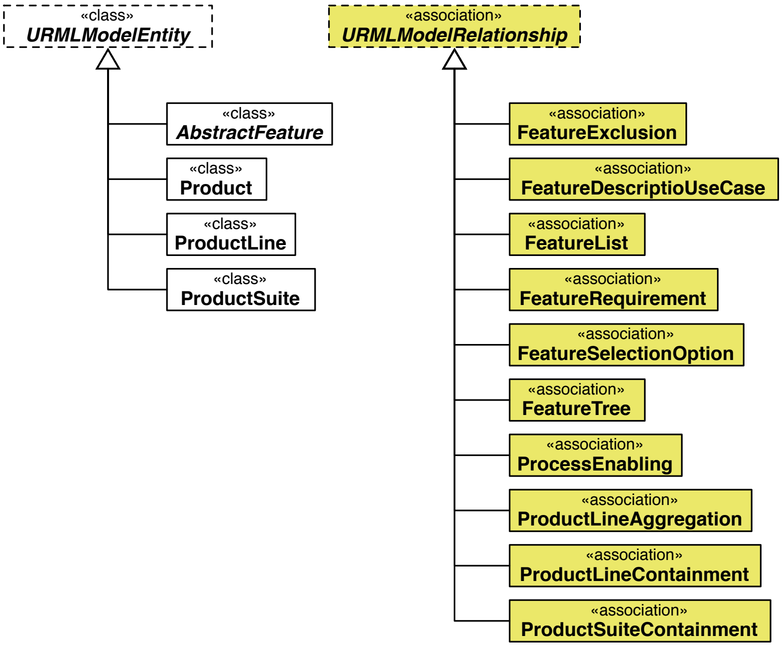


Figure 4.47.: Subclasses of URMLModelEntity and URMLModelRelationship, introduced in the Feature package.

This package has two main concerns: feature modeling and product line modeling. A Feature is a characteristic of a system that is visible to stakeholders. A Product has a set of Features, or in other words, has a FeatureList. Products can thus be compared in terms of their features.

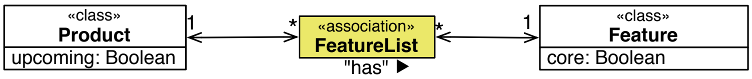


Figure 4.48.: A Product has no variability in it, it has a final list of Features.

A FeatureGroup represents a group of potential product features, called sub-features.

Which and how many of these sub features may end up in a product is governed by the selectionType attribute and the FeatureSelectionOption relationship. The selection type defines whether all, any, or exactly one of the sub features may be selected (expressed by the All, Any, and Exclusive literals of the FeatureSelectionType enumeration). The FeatureSelectionOption relationship defines which sub-features are part of the group and in addition can specify wether a certain sub-feature is mandatory. The FeatureSelectionOption relationship relates FeatureGroup to AbstractFeature, which together with its specialization relationships to Feature and FeatureGroup forms a composite pattern. Thus FeatureGroups can themselves contain FeatureGroups and a tree of features can be modeled. The root of such a tree is a FeatureGroup with the root attribute set tot true. For a part of the feature tree that is currently under construction, a feature group's underConstruction attribute can be set to true. That can facilitate partial reviews, where only the stable part of the feature model is put under inspection.

Apart from building a tree, AbstractFeatures may explicitly require other features being in the product (FeatureRequirement relationship), or exclude other features from the product (FeatureExclusion relationship). These relationships may relate features in different branches of the feature tree.[4]

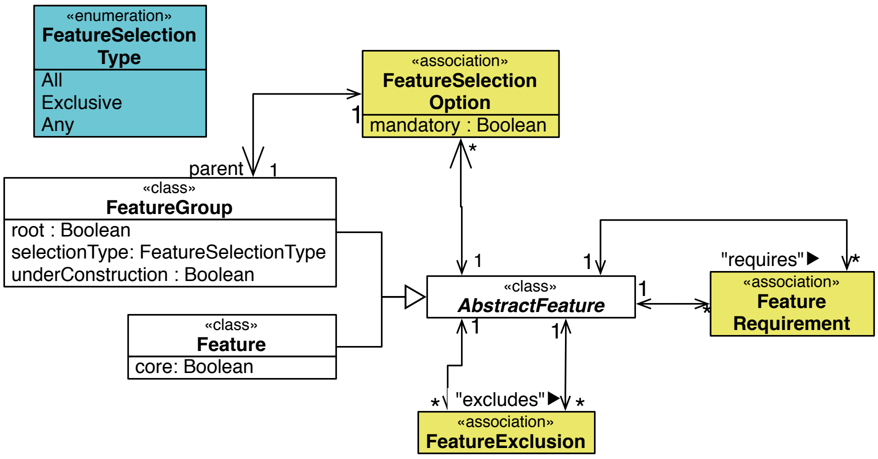
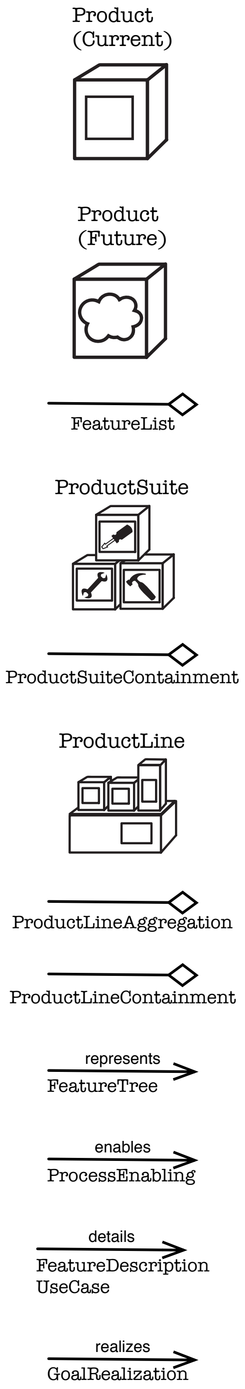


Figure 4.49.: The constituents of a feature tree.

A feature tree represents the features that products in a product line can have. [5]. Such a tree is rooted in a special instance of FeatureGroup, whose root attribute is set to true. There may be only one feature group with root set to true per product line.

[4] The feature tree is only a tree as long as we consider only the FeatureSelectionOption relationship. Adding instances of FeatureRequirement or FeatureExclusion makes it a directed graph.

[5] Called *feature tree* in product line engineering literature.



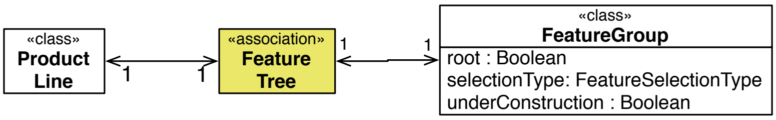


Figure 4.50.: A product has no variability in it, it has a final list of features.

Products are part of ProductSuites, which are collections of distinct products, related to the product with the ProductSuiteContainment relationship. Different from ProductSuites are ProductLines, from which products can be derived, or in other words, to which products belong (ProductLineContainment).

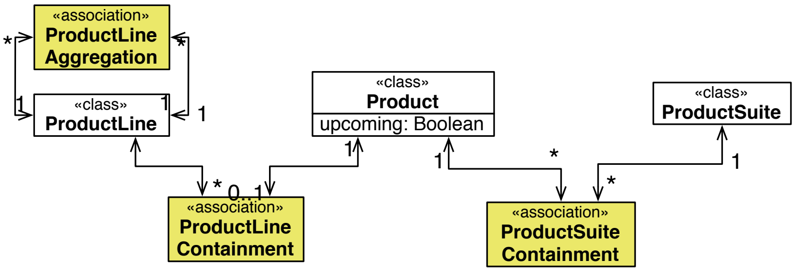


Figure 4.51.: How ProductLines relate to Products and to ProductSuites.

It can be shown how a feature is supported by a certain use case through the FeatureDescriptionUseCase relationship, which details how a feature is realized in terms of behavior of the system under discussion. Features of the system under discussion do enable processes in which the system under discussion is embedded. This can be modeled through the ProcessEnabling relationship.

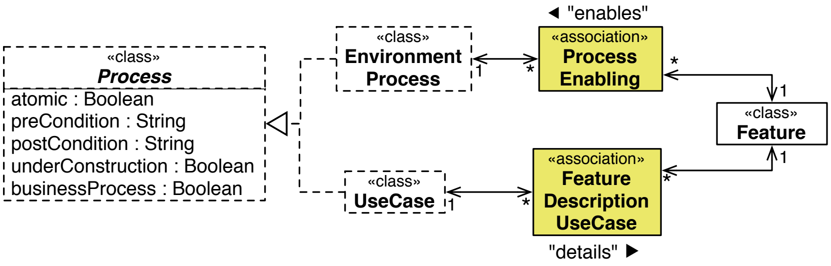


Figure 4.52.: A Feature can enable EnvironmentProcesses. The system function and behavior behind the Feature is detailed by UseCases.

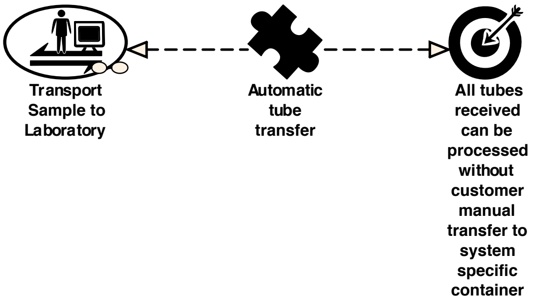

Transport Sample to Laboratory
Automatic tube transfer
All tubes received can be processed without customer manual transfer to system specific container

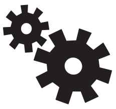
Figure 4.53.: One instance of `Feature`, related to an instance of `UseCase` by an instance of `FeatureDescriptionUseCase`, and an instance of `Goal` by an instance of `GoalRealization`.

In Figure 4.53, the automatic tube transfer feature is detailed by the workflow of the use case “transport sample to laboratory”.

A feature that is part of every product of a product line can be marked being a ’core’ feature in the feature tree. A product line can contain other product lines. A product can be marked as being an upcoming product, which means it is not being sold yet. For supporting review of feature models, feature groups can be marked being “under construction”. This is due to the fact that a feature group very likely has its own diagram. A feature group that is not under construction anymore has a reviewable diagram.

4.2.6. Requirements

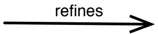
Functional Requirement



Functional Requirement (Regulatory)

Quality Requirement (Uncategorized)

Quality Requirement (Uncategorized, Regulatory)



RequirementRefinement

Quality Requirement (Efficiency)

Quality Requirement (Functionality)

The `Requirements` package introduces one meta-class that inherits from `URMLModelEntity` and six meta-classes that inherit form `URMLModelRelationship` (Fig. 4.54). `Requirement` inherits from `URMLModelEntity` and is further specialized within the package. `FeatureConstraint`, `FeatureDescriptionRequirement`, `ProcessConstraint`, `ProcessRequirement`, `RequestRealization`, and `RequirementRefinement` , inherit from `URMLModelRelationship`.

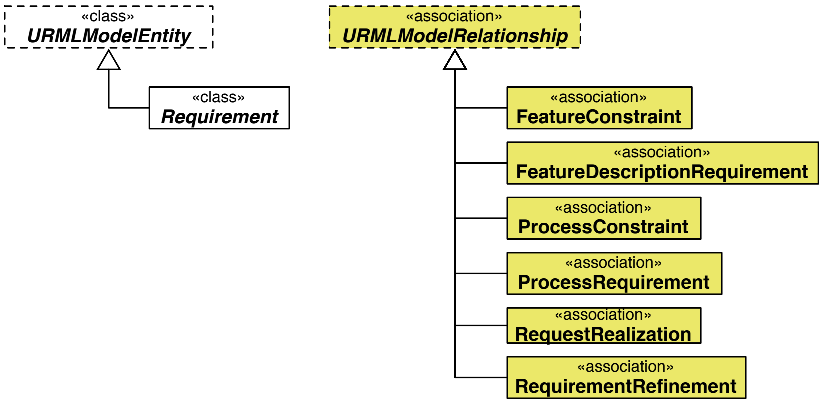


Figure 4.54.: Subclasses of `URMLModelEntity` and `URMLModelRelationship`, introduced in the `Requirements` package.

`Requirements` towards a system either ask for a functionality (`FunctionalRequirement`) or quality (`QualityRequirement`) of the system. Both types of requirement can be prioritized, ranked, marked being a cost driver, or marked as originating from a regulatory body. While two different requirements can have the same priority, they must not have the same rank.

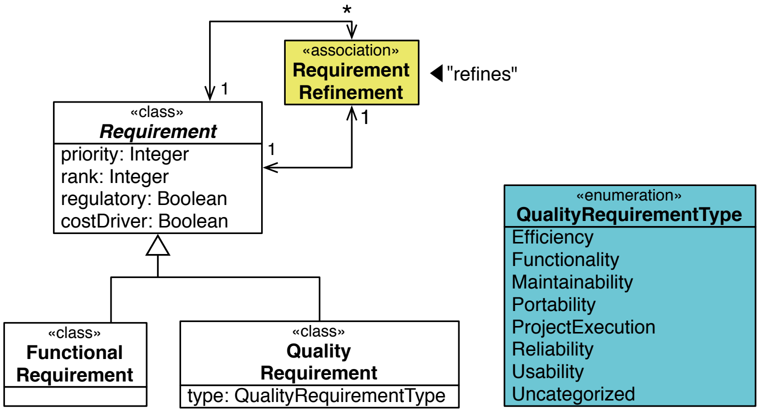


Figure 4.55.: `Requirements` are either regarding functions or quality of the system. `Requirements` can refine each other.

`QualityRequirements` can be further subcategorized via their `type` attribute. The according `QualityRequirementType` enumeration provides the possible values. All enumeration literals except the default value, `Uncategorized`, are taken from ISO 9126 (ISO 2001) (See Section A.4 for more details).

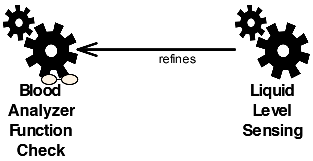


Figure 4.56.: Two instances of `FunctionalRequirement`, related by an instance of `RequirementRefinement`.

With the URML, the reason for the existence of requirements can be modeled. When the mitigation of a danger can be done inside the system, this can be expressed through a `RequirementMitigation` relationship.

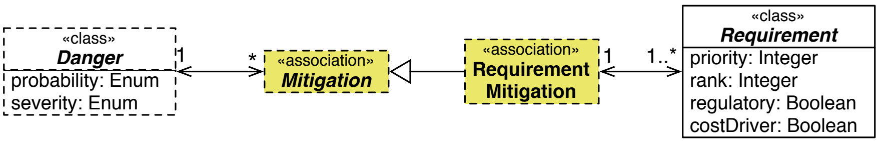


Figure 4.57.: A `Requirement` can mitigate a `Danger`.

`RequestRealization` allows to map requirements to stakeholder requests. This bands together what the stakeholder actually said with what the analyst derived from that statement.

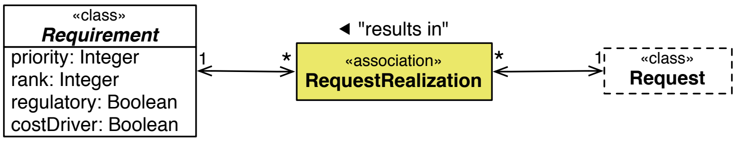


Figure 4.58.: A `Requirement` can realize stakeholder's `Requests`.

The combination of `RequestRealization and RequestCause` (the latter taken from the stakeholder and goal package) relationships offers one possibility to trace requirements to goals.

Quality Requirement (Maintainability)

Quality Requirement (Portability)

Quality Requirement (Project Execution)

Quality Requirement (Reliability)

Quality Requirement (Usability)

Quality Requirement (Efficiency, Regulatory)

Quality Requirement (Functional Suitability, Regulatory)

F

Quality Requirement (Maintainability, Regulatory)

M

Quality Requirement (Portability, Regulatory)

P

Quality Requirement (Project Execution, Regulatory)

Quality Requirement (Reliability, Regulatory)

R

Quality Requirement (Usability, Regulatory)

U

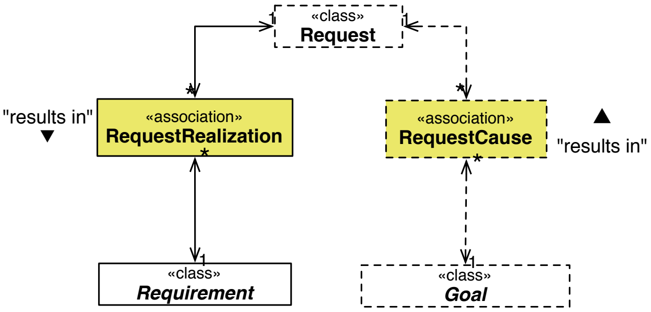


Figure 4.59.: `Requirements` can be traced to `Goals` via `RequestRealization` and `RequestCause` relationships.

Another possibility results from tracing requirements over features to goals.

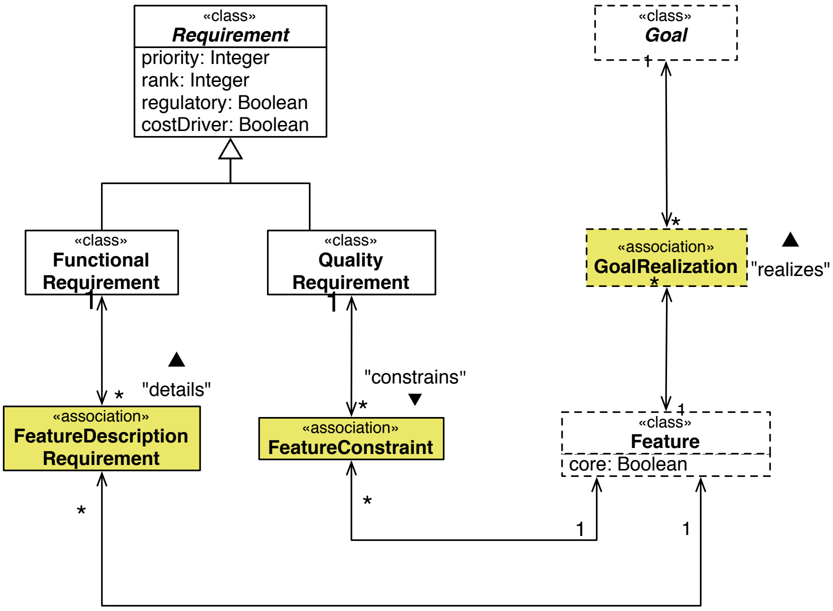


Figure 4.60.: `Requirements` can also be traced to `Goals` via `FeatureDescriptionRequirement` or `FeatureConstraint` relationships and then the `GoalRealization` relationship.

Functional requirements can be motivated by features (`FeatureDescriptionRequirement`) or describe constraints on features (`FeatureConstraint`).

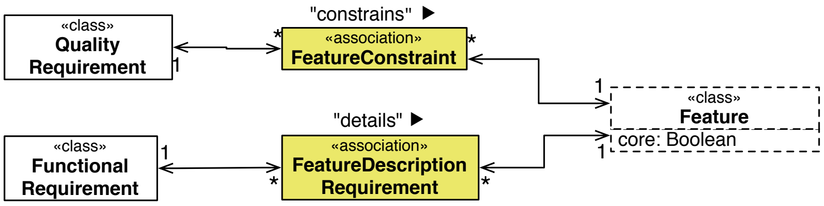
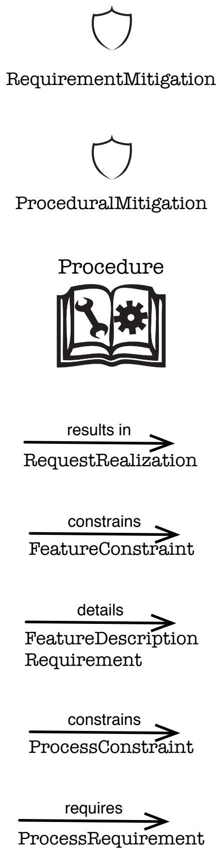


Figure 4.61.: A `QualityRequirement` can express a constraint on a `Feature`. A `FunctionalRequirement` provides details to a `Feature`.



It can be modeled how a process is constrained by a quality requirement (`ProcessConstraint`). Furthermore, a process can require a certain function of the system under discussion, which can be modeled through the `ProcessRequirement` relationship.

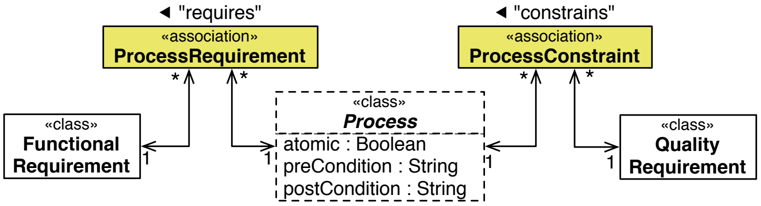


Figure 4.62.: A `QualityRequirement` can express a constraint on a `Process`. A `FunctionalRequirement` shows prerequisite functions of a `Process`.

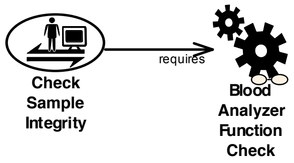


Figure 4.63.: An instance of `UseCase` related to an instance of `FunctionalRequirement` by an instance of `ProcessRequirement`.

4.2.7. Technical discussion of the abstract syntax meta-model

This subsection discusses technical peculiarities of the URML abstract syntax meta-model. As required, it is specified in MOF. As constraints on some meta-classes were required, the compliance level is CMOF. To the best of our knowledge, there is no implementation of a MOF compliance checker. Therefore, compliance to CMOF was ensured manually, based on checklists assembled from the MOF specification (OMG 2015b).

For enhancing readability of the meta-model, two notational conventions were followed that are not part of MOF. First, we have introduced keyword-like constructs that specify whether a given meta-model element has entity, enumeration, or relationship semantics. In addition, a color scheme is used that visually amplifies that differentiation: Entity meta-classes use a white background, relationship meta-classes a yellow background, and enumerations a turquoise background.

The meta-model sparsely employs multiple inheritance. Multiple inheritance is needed for the `Actor` class which inherits characteristics of `Stakeholder` and `System`. We have avoided deep inheritance hierarchies in all occasions where a subclass would have introduced no additional attributes. Instead, attributes typed by enumerations were introduced to allow for categorization of entity classes (e.g. `HazardType`, `ServiceProviderType`).

Relationship meta-classes are used to a large extent in URML. We have followed the approach that every meta-relationship having unique semantics should be represented by a relationship meta-class in the model. This was motivated by Olivé's definition of the term meta relationship type, which is "an entity type whose instances are both relationship types and entities" (Olivé 2007). Thus attributes of meta-relationships can be modeled without the use of association classes, which are not allowed by MOF, but within the first compartment of the class box. Examples of such attributes are the `description` attribute inherited by all subclasses of `URMLModelRelationship` and the `type` attributes of `GoalContribution`, `SystemInteraction`, or `InformationFlow`. In the case of the FeatureSelectionOption meta-relationship, the Using classes for modeling meta-relationships also enabled modeling of n-ary relationships. This would not be possible by using the `Association` meta-meta-class of MOF as it is required to be binary. For example, an instance of `RequirementMitigation` connects at least one instance of `Requirement`, one instance of a `Danger` mitigated, and one instance of a `HarmedElement` protected by the mitigation.

The abstract superclass of `RequirementMitigation`, `Mitigation`, is an interesting case. It is stating that any mitigation connects a `HarmedElement` and a `Danger`, but no mitigating element. The `URMLModelEntity` subclasses providing the mitigation are related to its concrete subclasses, `RequirementMitigation` and `ProceduralMitigation`. With this technique, it was not necessary to introduce another abstract class as a superclass of `Procedure` and `Requirement`, which would have increased the complexity of the meta-model.

4.3. URML Concrete Syntax

In the previous section, the abstract syntax of URML was explained, and the notation has only been introduced informally, wherever example model excerpts are presented. In this section, the concrete syntax model (CSM) of URML is described formally, and the architecture of the abstract-to-concrete-syntax mapping is outlined, both by means of a meta-model. To keep this section short, the detailed specification for every model element is not presented here, but can be found in the respective paragraphs of Section A.4. After the formal introduction of the notation, the development of the notational imagery is described, and then the notation is evaluated in terms the principles that were described in Section 2.4.

4.3.1. Concrete Syntax Model

The core of URML's concrete syntax model consists of three meta-classes and one enumeration. It is shown in Figure 4.64.

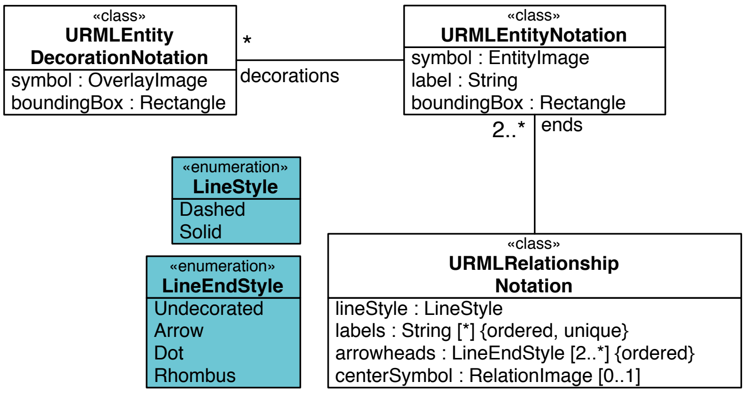


Figure 4.64.: Concrete Syntax Model of URML: Basic Structure

`URMLEntityNotation` reflects the structure of the notation of all entity meta-classes. Each entity meta-class is represented by a graphical `symbol` and a textual `label`. Optionally, some entity symbols can be decorated by smaller symbols, called overlays. The overlays are represented by the `URMLEntityDecorationNotation` meta-class. The concrete syntax model does not restrict the number of decorations, though there certainly is a natural limit imposed by human perception. If there are too many decorations, the overall symbol might be perceived as overcrowded. When this limit is reached, does however depend on factors external to the model and the concrete form of the base symbol. We have therefore not constrained the CSM to allow for extensibility. Figure 4.65 gives an example of a typical instantiation of `URMLEntityNotation` and `URMLEntityDecorationNotation`, to represent an instance of `FunctionalRequirement` with the `regulatory` attribute set to `true`.

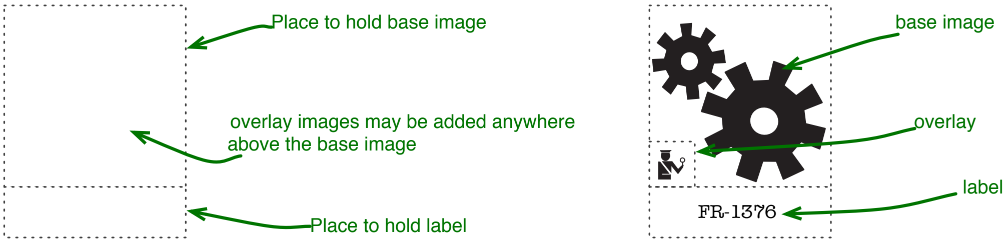


Figure 4.65.: Example of the visual structure of a URML entity notation

`URMLRelationshipNotation` realizes the visualization of relationship meta-classes. Each entity is visualized by at least one line, at least two arrowheads, and at least one label. Depending on the multiplicities of the relationship, more labels, arrowheads are needed, as well as a special relationship symbol for the which is the junction of all relationship lines. While the concrete syntax model does not prohibit the relationship symbol for binary relationships, it is typically not used for these. Again, the CSM was not constrained to allow for extensibility. Figure 4.66 gives an example of an instantiation of the model for the binary `SystemEmbedment` relationship.

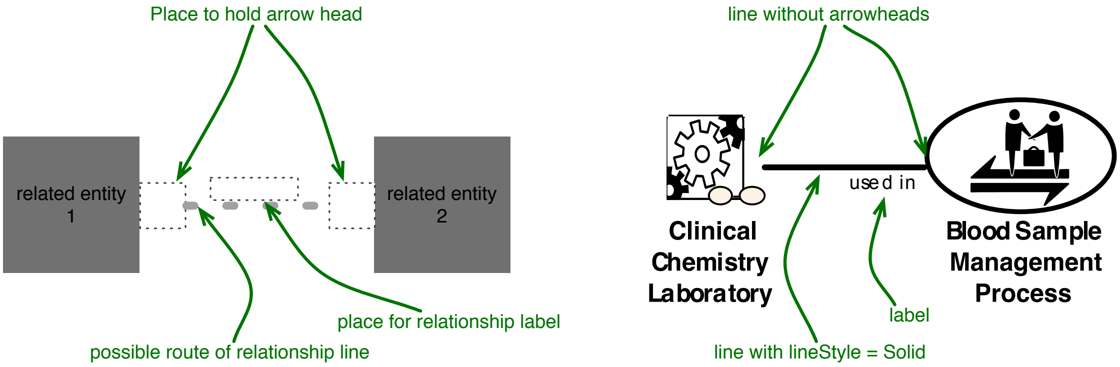

Place to hold arrow head
related entity 1
related entity 2
possible route of relationship line
place for relationship label
line without arrowheads
used in
Clinical Chemistry Laboratory
Blood Sample Management Process
label
line with lineStyle = Solid

Figure 4.66.: Example of the visual structure of a URML binary relationship notation

Figure 4.67 gives an example of the instantiation of the n-ary `SystemInteraction` relationship. If the notation for a binary relationship would define a relationship symbol, it would, analogously to Figure 4.67, be placed on the center of the line connecting the two entities. The placement of relationship labels in Figure 4.67 is in turn analogous to the placement in 4.67: The label should be somewhere near the line, and if possible not too close to the connected object, so that the relationship label can't be interpreted as the property of some connected entity.

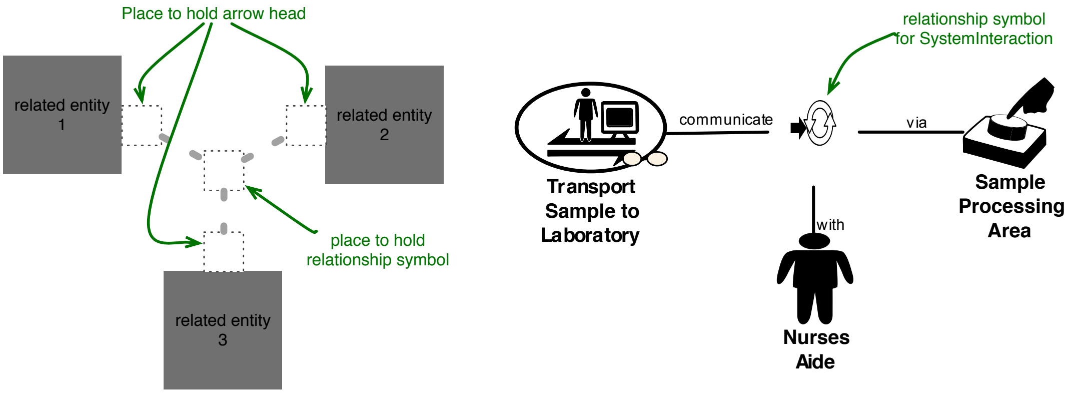


Figure 4.67.: Example of the visual structure of a URML ternary relationship notation

In Figure 4.64, some classes are only referred to as types of attributes, especially the meta-classes whose name ends with -Image. The top-level classes of these are shown in Figure 4.68. The concrete syntax of URML deals with four kinds of images: `EntityImages`, `OverlayImages`, `ArrowheadImages`, and `RelationImages`. Common to these abstract classes is the ability to refer to an image `location`. A complete list of subclasses is presented in Section A.5.

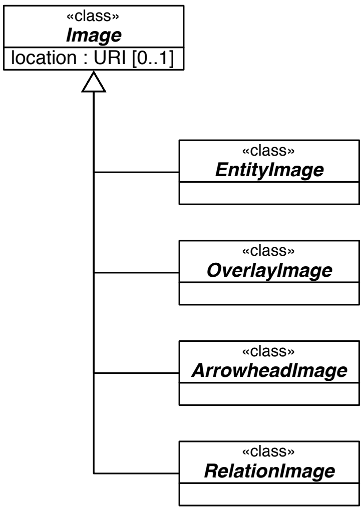


Figure 4.68.: Concrete Syntax Model of URML: Top level of image taxonomy

4.3.2. Mapping of Abstract to Concrete Syntax

In order to define which concrete syntax model elements are instantiated when an abstract syntax element is instantiated, a mapping of abstract to concrete syntax elements is needed. For this purpose another meta-model is created, that refers to elements of both abstract syntax model (ASM) and concrete syntax model (CSM). That additional meta-model is called ASM-CSM-mapping. For each instantiable abstract syntax model element, a display manager (DM) class is defined. Its associations to concrete syntax model elements define whether URMLEntityNotation or URMLRelationshipNotation will be instantiated. Constraints on the DM class define how exactly the according CSM element is instantiated, for example, where the text for a label comes from, or depending on which attributes of the ASM element images are chosen for the symbol.

The ASM-CSM-mapping of URML can be divided into two partitions: One partition is concerned with the notation for entity meta-classes, and the other partition is concerned with the notation for relationship meta-classes. The display managers for entity meta-classes all inherit from the abstract `URMLEntityDM`, which defines one constraint common to all these entity-related display managers. Every such display manager uses the `name` attribute of the `URMLModelEntity` subclass to define the value of the `label` attribute of `URMLEntityNotation`. This is shown in Figure 4.69

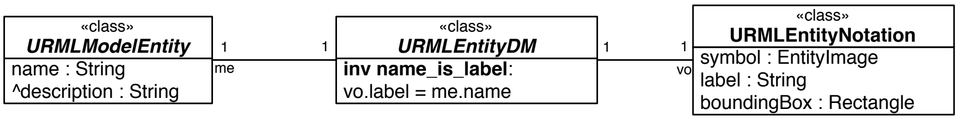


Figure 4.69.: ASM-CSM mapping for `URMLModelEntities`

The image to be used for the symbol attribute of `URMLEntityNotation` is then defined by the subclasses of `URMLEntityDM`. In the example below (Fig. 4.70), `ActorDM` defines that `ActorImage` is used on instantiation of `Actor`.

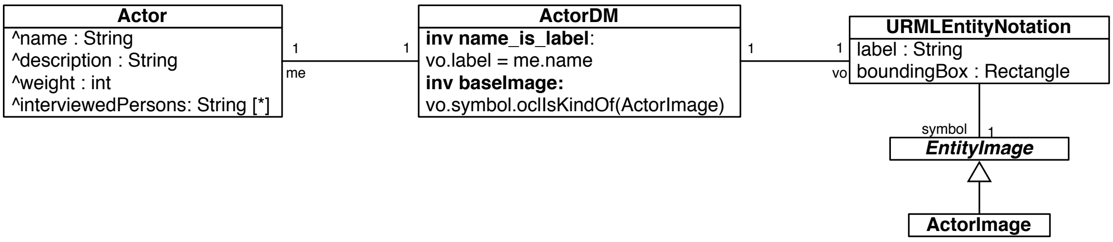


Figure 4.70.: ASM-CSM mapping for `Actor`

In some cases, the image used is depending on the value of a certain attribute. For example with `EntityObjects`, `EntityObjectDM` switches between `EntityObjectImageAtomic` and `-Composite`, depending on the value of the `atomic` attribute (See Fig. 4.71).

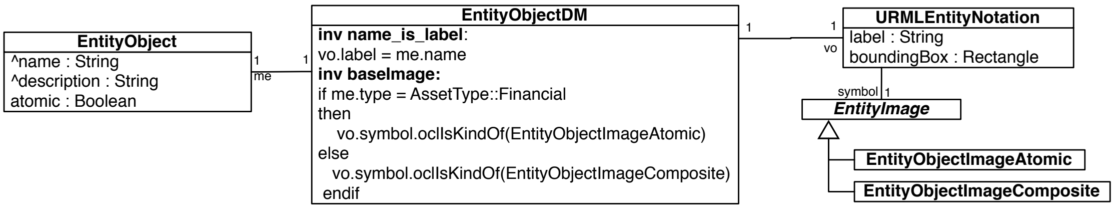


Figure 4.71.: ASM-CSM mapping for `EntityObject`

In addition to the base image, the symbol representing an element can also contain decorations. An example is the notation for `Feature` defined in Figure 4.72: `FeatureDM` uses the `core` attribute of `Feature` to define a constraint that requires or forbids the instantiation of the decorating `CoreOverlayImage`.

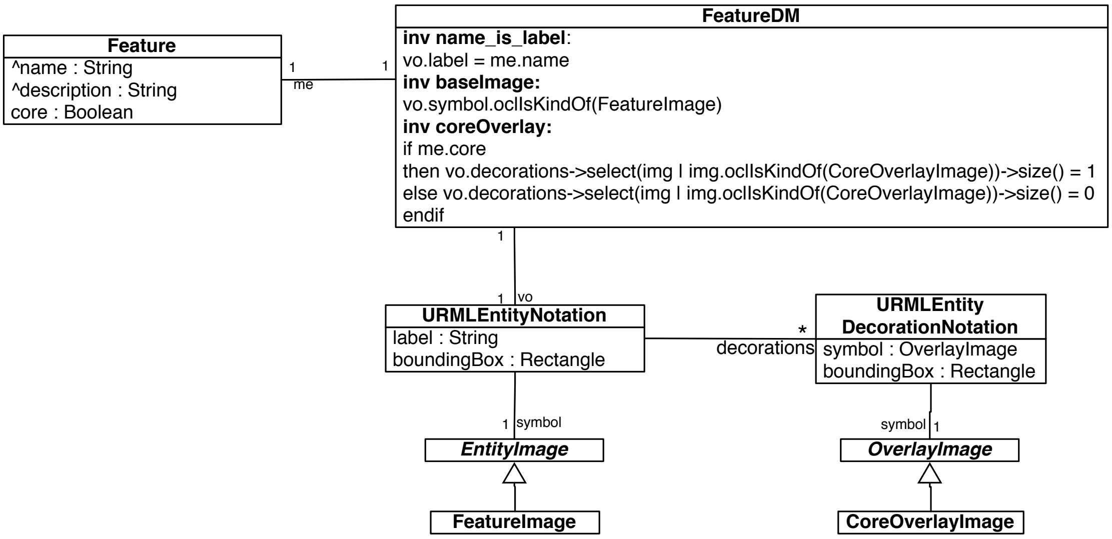


Figure 4.72.: ASM-CSM mapping for `EntityObject`

The cases discussed above already show the full complexity of the URML notation. The complete list of display manager classes is presented in Section A.4.

In the second partition of the ASM-CSM-mapping, which is concerned with relationship meta-classes, all display managers inherit from the abstract `URMLRelationshipDM`. Their sole commonality is the usage of the `URMLRelationshipNotation` meta-class.

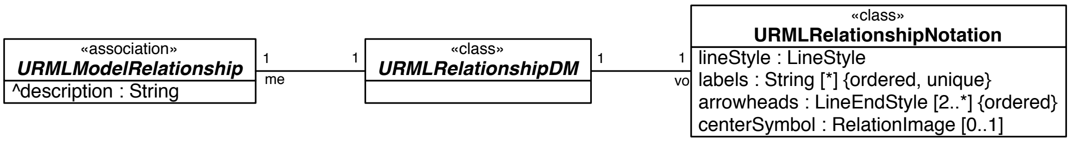


Figure 4.73.: ASM-CSM mapping for `URMLModelRelationships`

In URML we want the relationships to be labeled with static names, which means that every instance of a certain relationship meta-class will be accompanied by the same textual labels on a diagram. See for example Figure 4.66, where a `SystemEmbedment` relationship is instantiated. All instances of this relationship shall have a label with text ’used in’. This is similar to the ≪include≫ and ≪extend≫ keywords of UML that are placed on instances of `Include` and `Extend` relationships between `UseCase` instances. To the best of our knowledge, UML gives only an informal definition of the keywords, but not in terms of MOF, when defining the notation for `Include` and `Extend`. In a programming language, a static feature would be realized as a class variable. Class variables can be modeled in UML as static features (i.e. instances of Feature or a subclass with isStatic property set to true). However, MOF constraints disallow usage of that property. MOF offers an extension mechanism called `Tag`. A tag offers the possibility to add key-value pairs to an existing meta-class. Tags however have no defined concrete syntax. Therefore this mechanism was not used for URML’s concrete syntax model. Instead, the static label of a relationship is modeled as an invariant in the display manager classes corresponding to the respective `URMLModelRelationship` subclass. In Figure 4.74, an example is given: All instances of `AssetOwnership` have the same label ’has’.

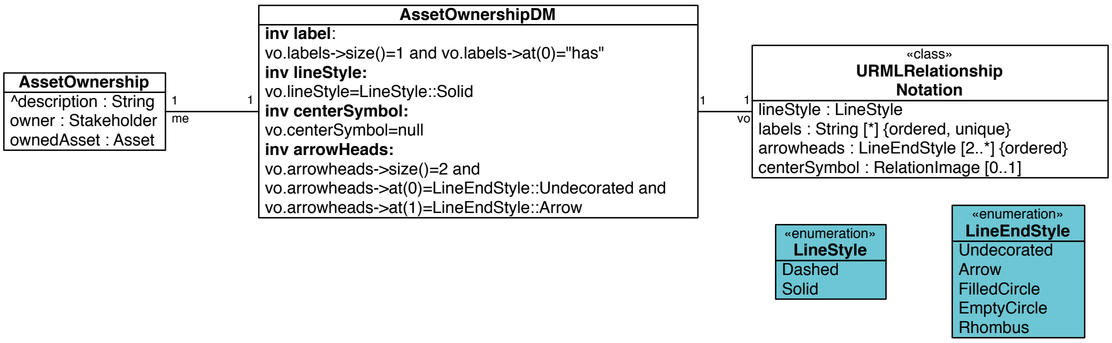


Figure 4.74.: ASM-CSM mapping for `AssetOwnership`

4.3.3. Implementation of Moody’s Principles

URML has a high *semiotic clarity*, as each symbol represents exactly one concrete meta-class. Some *symbol deficit* exists as the meta-class `URMLModel` has no symbol. Some attributes have no counterpart in the notation, as for example the `costDriver` attribute of `Requirement`.

We will discuss *perceptual discriminability* of the entity-related notation separately from the relationship-related notation. The relationship-related notation is based on arrows and labels. As the arrow lines of all relationship symbols have the same color, size, and texture, and we did not pre-scribe specific routing styles, we largely rely upon *textual differentiation*. Two exceptions from this rule are `Mitigation` and `SystemInteraction`, where a symbol is placed in the center of the n-ary relationship. For the entity notation, we achieved that almost every symbol has a unique shape. Only `UseCase` and `EnvironmentProcess` share the same shape, and the shape of the `CompositeEntityObject` symbol is very close in shape to that of `Product`. *Perceptual popout* is achieved by the iconic nature of the symbols: they do not only differ by shape, but also what is inside the shape. Other visual variables were not used.

The most important design goal apart from semiotic clarity and perceptual discriminability has been *semantic immediacy.* It has been difficult to achieve that for some abstractions. For these concepts it was hard to find an intuitive symbol, but we are confident that the general semantic translucency of URML is high.

URML supports inheritance and aggregation, and some special compositional relationships. These support *complexity management* in terms of the meta-model. Notationally, URML has also some support. For `Process`, we have designed an overlay to specify when the instance is not further decomposable. `Features` a different symbol than `FeatureGroups`, which are decomposable into `Features`. We also have designed a special symbol for the `FeatureGroup`, that roots the feature tree (i.e. its `root` attribute is set to true). `EntityObjects` have a different shape if the `atomic` attribute is set to true.

As can be seen in chapter 6, we borrowed a symbol from Enterprise Architecture CASE Tool (Sparx 2014) for the purpose of cognitive integration: whenever a symbol is linked to a diagram, a goggle symbol is displayed on the lower right corner of the symbol. If a diagram links to another diagram via a hyperlink, we have also relied upon built-in notation of the CASE tool.

4.4. Comparison to URN

URML and URN both focus on modeling requirements knowledge in general, and for both languages, it is claimed that they support early requirements elicitation. URN in addition supports discussing high-level architectural concerns, which is not a focus of URML. Regarding early requirements elicitation, URML and URN differ on the set of concepts needed to express requirements knowledge. URN focuses mostly on goal-oriented concepts with GRL. UCM has to fulfill a dual role. It is used for expressing functional requirements as well as high-level design. URML also supports modeling of hard goals, soft goals, goal operationalizations, and workflows. But URML offers a wider range of concepts for managing early requirements knowledge. In addition it facilitates modeling product lines with features, early safety engineering with hazards and early security engineering with threats. URML supports modeling interaction with other systems via boundary objects and modeling internal black box components with service providers. URML not only models actors, but two additional types of stakeholder: customers and business stakeholders. URML supports quick data entry during stakeholder interviews by a generic stakeholder request abstraction and innovation modeling an abstraction for capturing ideas. In contrast to URN, URML explicitly contains a requirement abstraction and taxonomy in its meta-model and allows distinguishing regulatory requirements from other requirements. Regarding workflow modeling, URML is less detailed regarding the exact nature of the workflows. URML also allows to model workflows (or scenarios, or processes) on a high level via `EnvironmentProcess` and `UseCase`, and allows for an ordering of workflow steps via the `ProcessPrecedence` relationship. But it does not provide the capability to show more details of the workflow. UCM in contrast has a big set of PathNode specializations to show concurrency, decisions, timed events, or dynamic structuring. The decompositionType attribute of IntentionalElement is similar to the selectionType of FeatureGroup in URML. The difference is that URN discusses the variability in goals whereas URML discusses the variability in features.

Table 4.1 shows which abstractions of URN can be mapped to abstractions of URML. The equivalence column suggests how close the abstractions are in terms of semantics. Based on the table, partial translation of models between the two languages is possible. Moreover, the abstraction equivalencies can be the base of suggestions for traceability links between URN and URML models. For example, URN IntentionalElement instances with type Goal could be translated 1:1 into URML HardGoal instances. Other concepts do not directly correspond but have, to a certain degree equivalent meaning. For example, Responsibility instances on UCM diagrams could be candidates for being translated into FunctionalRequirements or UseCases. The modeler has to choose based on the granularity of the Responsibility instances. FailurePoint instances in UCM diagrams could be interpreted in some rare cases as URML Vulnerabilities and more often as ProcessExtension. In both cases, the name of the failure point would become the name of the linked Danger or Process instance.

Table 4.1.: Related abstractions in URN and URML

URN abstraction	**URML abstraction**	**Equivalence**
IntentionalElement, type = Goal	HardGoal	Strong
IntentionalElement, type = SoftGoal	SoftGoal	Strong
IntentionalElement, type = Resource	EntityObject	Strong
IntentionalElement, type = Belief	Idea	Weak
IntentionalElement, type = Task	EnvironmentProcess, UseCase	Weak - Medium
Actor	Actor, Stakeholder, ServiceProvider	Medium
Responsibility	FunctionalRequirement, UseCase	Medium
Decomposition	GoalDecomposition	Strong
Contribution	GoalContribution	Strong
NodeConnection	ProcessPrecedence	Weak
Stub	ProcessInclusion, ProcessExtension, Feature	Weak
FailurePoint	ProcessExtension, Vulnerability	Weak

4.5. Comparison to KAOS

KAOS is predominantly a goal-oriented requirements modeling language. This means that the Goal abstraction plays a central role in the meta-model, and that requirements knowledge encoded in KAOS models is very much shaped by the analyst looking at the requirements from a goal-oriented perspective. That does not mean that other views cannot be modeled with KAOS. Beyond the intentional view, KAOS supports modeling risk with Obstacles, structure with Objects, usage with Scenarios and EnvironmentAgents, internal behavior with StateMachines. As KAOS supports risk modeling, it is a little closer to URML than URN is. URML does not support to express the same amount of detail that KAOS supports with state machine diagrams and scenario charts. It is also a little less detailed on the modeling of software objects. KAOS does not have dedicated abstractions for feature, use case, and stakeholder modeling. Therefore URML offers greater flexibility to the modeler regarding how to encode the requirements knowledge, and also how to structure the elicitation process. With KAOS, the modeler may run into problems early on if the stakeholders being interviewed do not appreciate goal modeling.

Table 4.2 relates KAOS and URML abstractions. We have assessed five abstractions to have a strong equivalency, so they can be easily translated between the languages, e.g. SoftGoals. Others are not so straightforward to map. For example KAOS BehavioralGoal instances may in some cases be translated to URML HardGoal instances. In some other cases, the URML modeler might want to avoid deep goal hierarchies, as part of the knowledge is already encoded in instances of UseCase. While UseCase has very different semantics from Goal, it appears that some BehavioralGoal instances would, with slight revision, provide the name for a UseCase. This has also been suggested by Cockburn (Cockburn 2000). An example of a weak equivalence is the Operationalization relationship that connects Requirements to Operations. It cannot directly be translated from KAOS to URML, but for a given Operationalization in a KAOS model, there should be a corresponding GoalRealization in a mapped URML model.

Table 4.2.: Related abstractions in KAOS and URML

KAOS abstraction	URML abstraction	Equivalence
SoftGoal	SoftGoal	Strong
BehavioralGoal	HardGoal	Medium
Requirement	Requirement	Strong
Software-To-Be-Agent	ServiceProvider, SystemUnderDiscussion	Medium
EnvironmentAgent	Actor	Strong
Operation	UseCase	Medium
Obstacle	Danger	Strong
BoundaryCondition	ProcessExtension, Vulnerability, DangerTrigger	Medium

Resolution	RequirementMitigation, ProcessExtension	Weak-Medium
sysID attribute in model annotations	Product	Strong
Operationalization	GoalRealization	Weak
Expectation	EnvironmentProcess	Weak
Entity	EntityObject	Strong
Refinement	GoalDecomposition, GoalContribution, RequirementRefinement	Weak

5. Towards a framework for model-driven elicitation strategies

The abstract syntax meta-model of URML can be used as a guideline how to perform requirements elicitation, with the primary goal to populate and then consolidate an early requirements model. In order to arrive at a satisfying requirements model, a series of questions are posed to different stakeholders, and the answers are transformed into model elements. Whenever new elements are added to the model, they have to be aligned to the knowledge already encoded in the model. Usually the whole process is done informally, based on the intuition of the requirements analyst, which is to a certain degree acquainted with the requirements modeling language of his choice. However, strategies how to traverse the meta-model in order to populate the requirements model can be explicitly specified and reused. Only in KAOS we can already see an effort towards explicitly describing elicitation strategies and tactics, as introduced in Section 3.5. This chapter outlines the vision of a standard framework for that purpose. That framework is independent of a concrete modeling language - it shall facilitate the expression of strategies to make strategies using different languages comparable. The framework is about creating questions based on the language meta-model and the current state of a concrete requirements model. These questions can be used in interviews with stakeholders, could be exported to documents, or be posed interactively in a CASE tool using the framework.

Section 5.1 describes tactics and operations used to prepare question catalogs for the requirements elicitation phase. Section 5.2 describes checks for model quality evaluation, that can be used independently of the question generator, but also provide input to elicitation strategies. Section 5.3 then describes generic elicitation strategies.

5.1. Constituents of an Elicitation Strategy

As described in section 2.5.2, Dardenne et al. already stated that the meta-model guides the definition of “acquisition strategies” (Dardenne, Lamsweerde, and Fickas 1993, p.7). We use the term *elicitation strategy* in this section. An elicitation strategy provides rules that govern which questions are asked in which order.

The constituents of a strategy are called *elicitation tactics*. An elicitation strategy instantiates and combines several tactics. Depending on the experience of the requirements analyst, time constraints, involvement of stakeholders, different tactics are selected. A tactic defines with which questions to start, how to proceed after an answer was given or if no answer is given, how long to pose the same kinds of questions, and when to switch

to a different kind of question.
For executing an elicitation tactic, a series of *elicitation operations* need to performed. Elicitation operations work on the requirements model and the meta-model. For example, an operation can take a given model element as input and determine its type, i.e. the meta-class that it instantiates. Another operation could take that meta-class as an input and deliver the meta-relationships to the meta-class. By executing a series of operations, a tactic generates a *path* through model and meta-model elements. This path is then translated into an *elicitation question.*
The approach described above is supported by two components, the *question generator* and the *question concretizer.* Given a specific starting point in the meta-model, and a given tactic, the question generator generates a set of questions. These questions are the basis of an elicitation session to be answered by a stakeholder. The answers are taken as new input to populate the requirements model. Questions using only the meta-model's terms are abstract, as the vocabulary used is based on meta-class' names. For example, "What actors does the system have?" is more abstract than asking "Will Jim use the new system?". This abstractness cannot be avoided for an empty meta-model.

The operations described in this section are the building blocks from which a tactic can be built. Some operations work on the model to deliver information about the contents of the model. Some operations work on the meta-model to structure the model traversal. Some operations work on both model and meta-model. Performing the aforementioned operations in sequence allows us to construct paths. The basic idea of the question generator is that a question regarding something in the model can be considered as a path through the model, and inversely, that any path through the model can be translated into one or many questions. Therefore, operations to translate a path into a questions are needed. In the following paragraphs, we will first explain the model and meta-model and operations, before going into the translation operations.

As soon as there are artifacts in the requirements model, the artifacts in the meta-model can be used to concretize the abstract questions generated by the question generator. The meta-model prescribes what queries can be made on an existing model. For example, for a given part of a path constructed by the generator, query the model if it already contains artifacts that match the path. This information can be used to concretize the abstract questions. The component querying an existing requirements model is called *question concretizer.* A framework consisting of "question generator" and "question concretizer" supports the requirements analyst throughout the early requirements engineering process. Starting with some abstract questions, the model is initialized from the answers to these questions. From the model elements in the incomplete model, further questions can be constructed, gradually becoming more concrete, the more elements are already available in the model.

5.1.1. Operations

The operations upon which the tactics are based take either a model or meta-model element and provide a primitive type, a model element, or a meta-model element in return. Many of the operations working on model and meta-model would be supported by a programming language that offers a reflection API. The operations returning a primitive type either compute a metric or are about translating model or meta-model contents into strings containing natural language.

`GetSuperClass` takes a meta-class and returns its subclass if it has one. `GetSubclasses` works in the opposite direction. `GetMetaRelationships` returns the relationship meta-classes of an entity meta-class. `GetNumberOfMetaRelationships` uses `GetMetaRelationships` and counts the elements of its result set. `GetOppositesFromRelationship` takes a meta-relationship and a meta-class and delivers the other participants of the relationship.

`GetMetaClass` takes a model element as an input and returns the meta-class that it instantiates. `GetInstancesOfMetaClass` takes a meta-class and delivers the set of model elements that are an instance of it.

`GetRelationships` takes a model element and delivers the instances of relationship meta-classes in which the element participates. `GetRelatedInstances` takes a model element, uses the `GetRelationships` operation to collect the relationships that it participates in, and for each relationship gets the other participants of the relationship. `GetUninstantiatedRelationships` delivers the meta-classes of relationships a model element could participate in, but were not instantiated yet in the model. It takes a model element, uses `GetMetaClass` to determine its meta-class, `GetMetaRelationships` to gather the the relationships it could theoretically participate in, and from this set removes the meta-classes obtained by using `GetRelationships` and `GetMetaClass`. `GetAttributeValue` takes a model element and an attribute name to deliver the value of the given attribute.

To translate paths to questions, we need the following operations: `TranslateMetaEntity` returns a natural language term for a given entity meta-class, which is a noun. An additional parameter determines whether the returned noun is in singular or plural form. `TranslateMetaRelationship` returns a partial verb phrase, with placeholders where the nouns representing the related entity meta-classes need to be filled in. `TranslateEntity` returns the value of the name attribute of the given instance. `TranslateRelationship` creates an analogy question.[1]

Figure 5.1 gives two examples of what is to be achieved by a combination of such operations. The setting of both examples is a URML requirements model for an automated teller machine (ATM). In the first example, an instance of `Actor` named `BankCustomer` was picked as a starting point. Via operations on the meta-model it is detected that `Actor` participates in the `SystemInteraction` relationship, connecting `Actor` to `UseCase` and `BoundaryObject`. Via operations on the model it is detected that an instance of `UseCase` named `DrawMoney` already is in the model. The requirements analyst decides to use that use case as an additional input to the question generation. Finally the framework generates the question presented in the figure. In the second example, the starting point is an instance of `Threat`. This time, no additional elements from the model are considered, either because no instance of `UseCase` is present yet or the analyst

[1] As relationship instances have no name, the simplest but useless option would be to let it return the same as TranslateMetaRelationship.

has chosen not to take any into account.

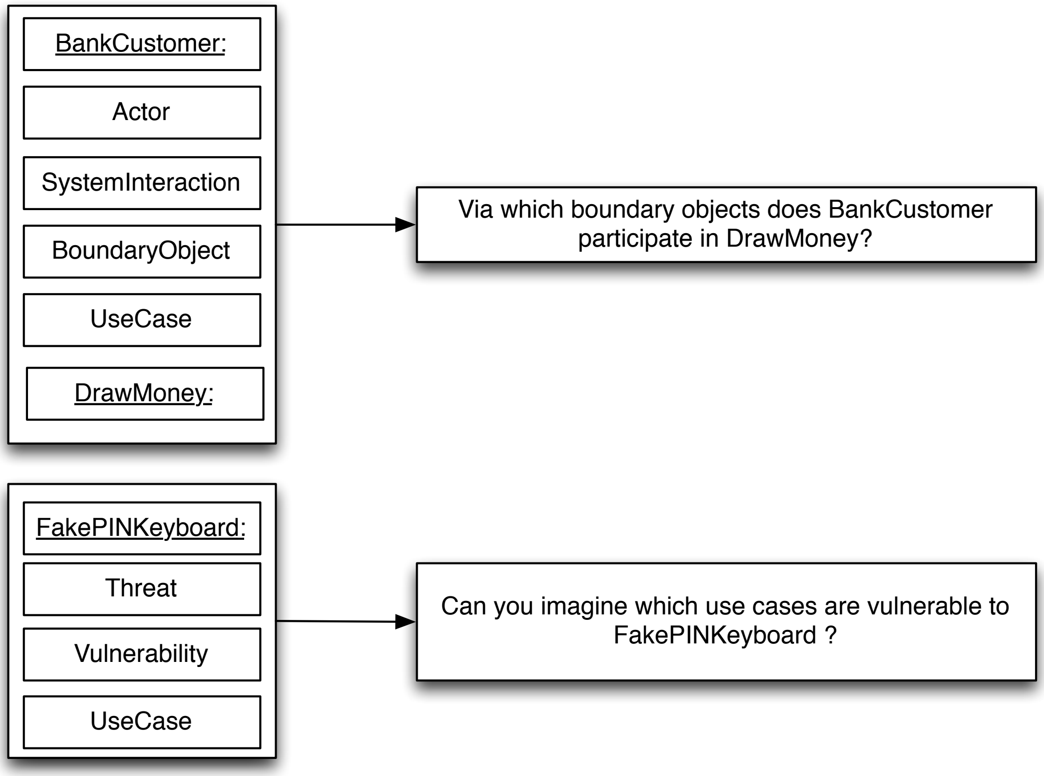

BankCustomer:
Actor
SystemInteraction
BoundaryObject
UseCase
DrawMoney:
Via which boundary objects does BankCustomer participate in DrawMoney?
FakePINKeyboard:
Threat
Vulnerability
UseCase
Can you imagine which use cases are vulnerable to FakePINKeyboard ?

Figure 5.1.: Question generation examples

The first example in Figure 5.1 has already been an example of question concretization. If the analyst had not decided to include `DrawMoney` into the generation, the question would have become 'Via which boundary objects does BankCustomer participate in use cases?'.

Another possibility of concretization is based on the attributes of the relationship. The `SystemInteraction` relationship for example has a `type` attribute with the possible values `Initiation` or `Participation`, with Participation being the default value. As an additional concretization, the analyst could have decided to directly consider that aspect of `SystemInteraction`, so the question would have been 'Via which boundary objects does BankCustomer initiate use cases?' if the use case instance was not taken into account or 'Via which boundary object does BankCustomer initiate DrawMoney?'.

In the context of question concretization, the high-level goal to generate questions for stakeholders, is refined to the goal of generating questions with as less technical terms of the meta-model as possible. Thus, it should be avoided to refer to abstract classes in questions. Abstract classes are merely a structuring mechanism on the meta-model and are usually not visible to users of a graphical modeling language. Asking for these in an elicitation session will be counterproductive. E.g. “What HarmedElements does this Danger have?” is much harder to understand than “What Assets does this Danger harm?”. As a resolution to this problem, we proceed as follows. For a source-meta-class, we also consider relationships of its super classes as valid path components. For target-meta-classes, we don’t count inheritance relationships being extra path components. Thus the path `(Actor->isA->Stakeholder)->GoalStatement->(Goal->super->HardGoal)` is considered to be a path from `Actor` to `HardGoal` with three components.

The referral to concrete meta-classes in questions may in some occasions be unavoidable, if the type of the element is not deducible from its name, or apparent from the context of the elicitation sessions. For instance, instead of referring to DrawMoney, the question can use the phrase ’the DrawMoney use case’. In future work, the question generation can be improved by having the abstract syntax meta-model support the specification of natural language synonyms for given elements of a requirements model. The vision is that the question generator could generate the question ’Via which interface elements of the ATM does the bank customer draw money’ instead of ’Via which boundary object does BankCustomer imitate DrawMoney?’.

5.1.2. Tactics

This section describes elicitation tactics. The purpose of any tactic is to generate a set of questions. The answers to the questions can be interpreted by the requirements analyst to refine and populate the requirements model. The different tactics differ in how they traverse the requirements model and the language meta-model. A tactic describes how an algorithm starting at a source-meta-class can arrive at a certain target-meta-class, building a path of meta-classes and -relationships between the two. Any tactic is built upon the operations described in the previous section.

Figure 5.2 provides an example how a tactic is used within a strategy, how it uses operations to transform knowledge about the meta-model into an elicitation question, and how a stakeholder’s answer is translated into an element of the requirements model.

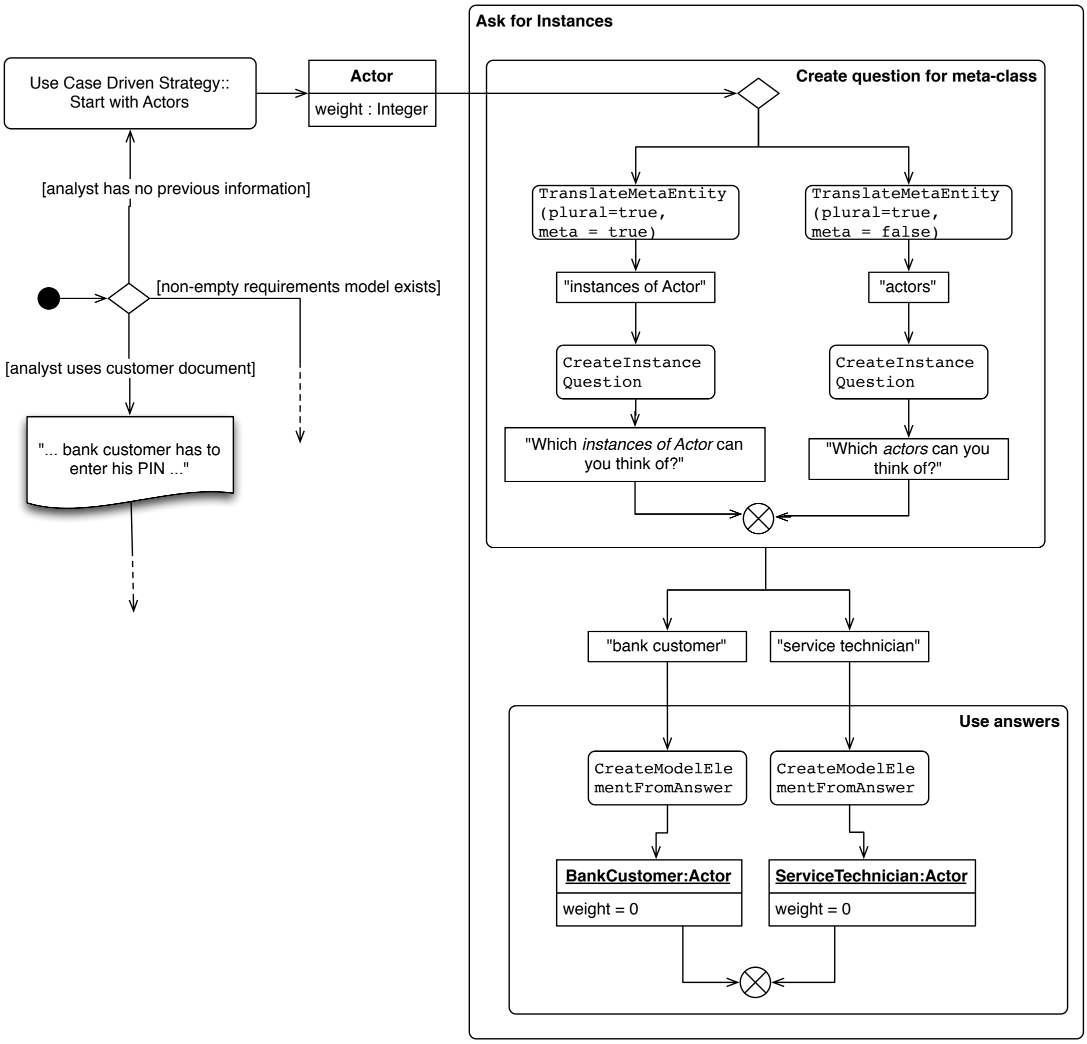


Figure 5.2.: Question generation example - Collect Entities

Collect Entities For an input set of entity meta-classes, ask the stakeholder for instances he is aware of. This tactic is a rather simple one, but effective if the stakeholder is acquainted at least partially with the terminology of the modeling language.

Missing Entities First This tactic is a variant of Collect Entities that specifically defines the input set of entity meta-classes. For every concrete entity meta-class, query the model if instances of the class exist (`GetInstancesOfMetaClass`) and filter those for which no instances exist. For each non-instantiated meta-class, perform `Collect Entities`.

Relationship-Driven Some stakeholders might prefer to talk about relationships instead of entities which are put into relationships later. This tactic generates a question for every relationship meta-class of the meta-model, or for a subset defined by parameters. As relationships do not exist on their own, this tactic will elicit (entity, relationship, entity)-tuples, which might be a quicker way of populating the model than Collect Entities.

Relationship Analogy For any given instantiated relationship, analogy questions are generated that ask whether similar relationships exist. For binary relationships, three questions would be generated:

1. Does source also have such a relationship to other targets?
2. Does target also have such a relationship to other sources?
3. Can you imagine such relationships for other target-source pairs?

This can be done analogously for n-ary relationships. The tactic requires an already populated requirements model.

Direct Neighbors For a given model element, this tactic determines its meta-class and traverses the meta-model to find concrete classes that are its *direct neighbors*. A direct neighbor is another meta-class between which and the source meta-class exists a meta-relationship. The traversal algorithm stops after having visited one concrete relationship and one concrete class. The tactic stops when all neighbors have been found. It uses the operations `GetMetaClass`, `GetSuperClass`, `GetInterfaces`, `GetMetaRelationships`, `GetOppositesFromRelationship`, and `GetSubclasses`. Figure 5.3shows the resulting paths of the tactic, when starting with an `Actor` (name : BankCustomer). The initial questions generated from these paths will be half concrete and half abstract. For example, the path to `Threat` via `Harm` can translate to: ≪Which threats are you aware of that can harm BankCustomer?≫ If the model already contains `Threats` that are unrelated to BankCustomer, we can use these to ask questions like ≪Is StolenCreditCard a threat that can harm BankCustomer?≫. If the model contains no such `Threats`, but only `Threats` that are already related to BankCustomer, the requirements analyst has to decide whether this is a criterion to stop using the given path. Alternatively he can still proceed by varying the initial question with a prefix sentence: ≪LeakedPIN is a threat that can harm BankCustomer. Can you imagine other threats?≫.

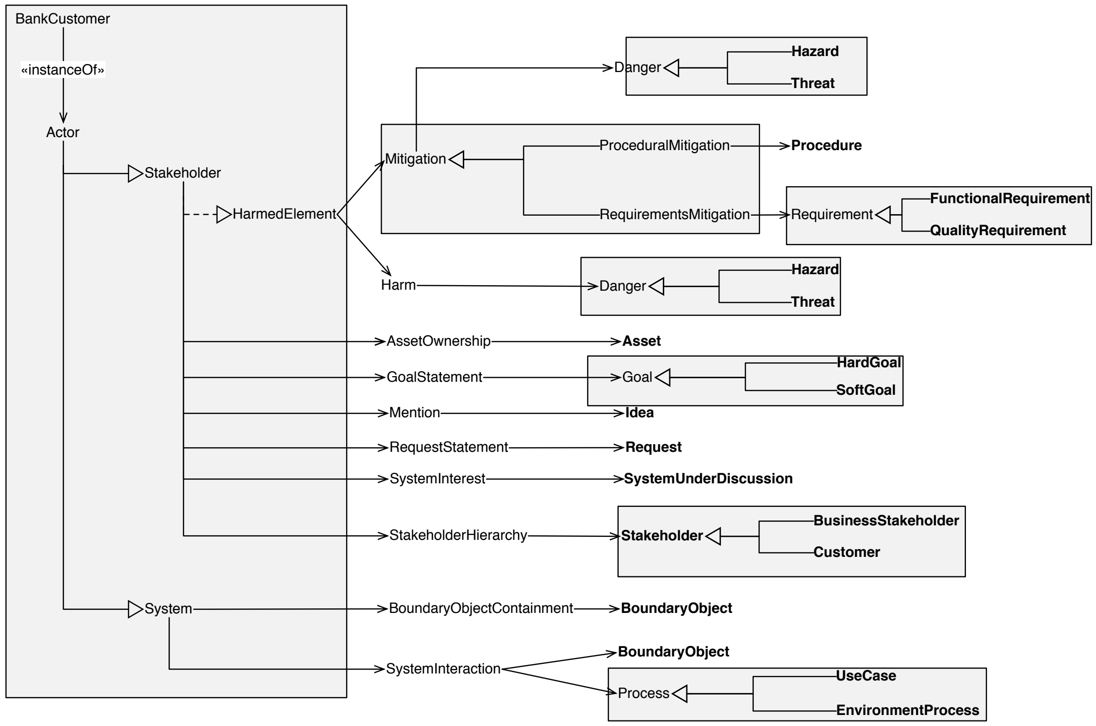


Figure 5.3.: The abstract paths starting from Actor, constructed with Direct Neighbors.

Specific Neighbor For a given model element and entity meta-class, obtain the meta-relationships connecting the given model element's meta-class and the given entity meta-class. The precondition that the given meta-class actually is a direct neighbor of the given model element's meta-class has to hold for this tactic. This tactic is useful if a using strategy prescribes a dedicated order of meta-classes in which the meta-model shall be traversed.

For example, a strategy could prescribe that when the analyst decides that enough goals have been identified, the next step should be to identify features that can be mapped to the goals. In contrast, an application of Direct Neighbors in this situation would create questions about stakeholders, requests, assessment sketches, and features.

Self-Relationships Self-Relationships is a tactic that creates a subset of the paths found by Direct Neighbors. For a given meta-class, it returns paths that contain exactly one relationship meta-class that leads to the input meta-class again. It is different from Direct Neighbors as its aim is to create more instances of the same meta-class. As an example, it could be part of a strategy to create a hierarchy of use cases first before discussing Actors, BoundaryObjects and Requirements. Such a strategy can use the Self-Relationships Tactic.

In URML, the following relationships connect instances of the same meta-class: StakeholderHierarchy, GoalDecomposition, GoalContribution, ProcessInclusion, ProcessExtension, ProcessPrecedence, Requirement Refinement, ProductLineAggregation, FeatureSelectionOption, FeatureExclusion, FeatureRequirement, Inheritance, Aggregation. This means the tactic is particularly suitable when elicitation is about Processes and Features (four self-referencing relationships), as well as Stakeholders and Goals (three self-referencing relationships). All other URML entity meta-classes have only the two self-referencing relationships inherited from URMLModelEntity, Inheritance and Aggregation. In these special cases, the tactic should not be used too long.

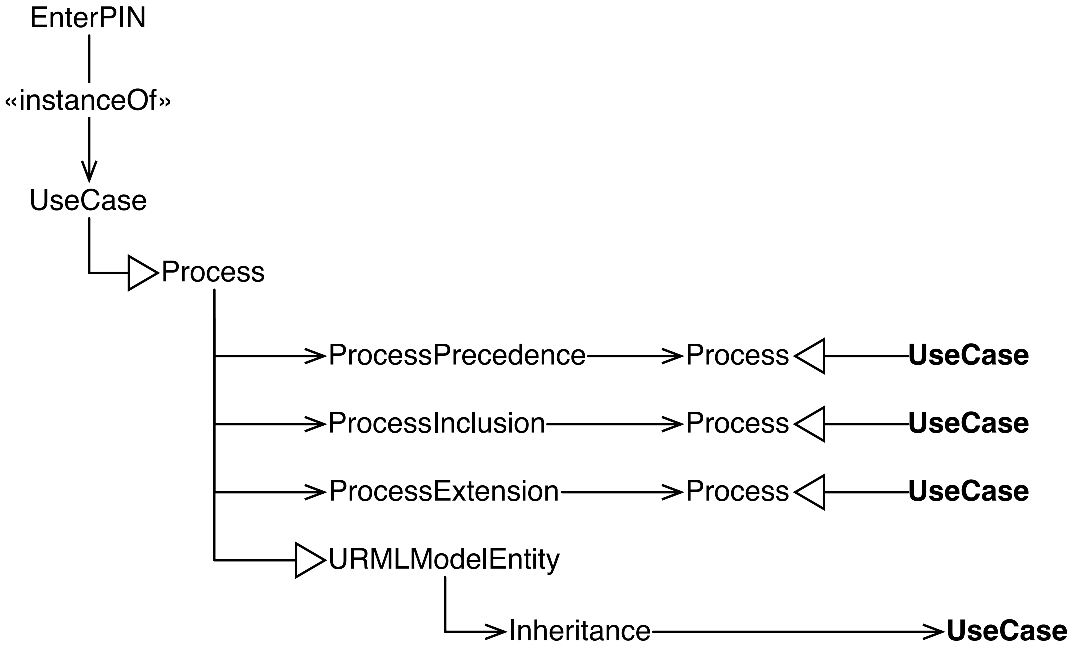


Figure 5.4.: The abstract paths starting from UseCase, constructed with Self-Relationships.

Source-Target Instead of focusing on the context of a model element, the meta-model can be traversed in a depth-first fashion. One example of this is the source-target tactic. It searches for all cycle-free paths between two meta-classes. The traversal algorithm for one path stops upon finding the target meta-class (success) or fails because he encounters a concrete class again that was already visited (cycle) or because the meta-class at the end of the path has no more associations to follow (dead-end). This tactic can potentially generate very long paths, which when translated to a single question can be very cumbersome to ask. It is therefore recommended to construct questions as in the direct neighbors tactic, but continuing with the next path component on the given path as soon as a new model element was specified. If a stakeholder digresses by providing multiple answers to the same question, the requirements analyst should continue the questions with only one of the answers. Optionally he can return to that point to perform the partial path with the other answers before continuing with the next path. Alternatively the additional answers are only recorded and will be used in other tactics after the current has been completely executed. When executing source target, the requirements analyst should experiment with setting a maximum path length as an additional parameter. As the paths can have many components before a traversal ends at the target class the probability is high that many paths share multiple components. Alternatively, the requirements analyst can decide not to use all the paths generated by this tactic.

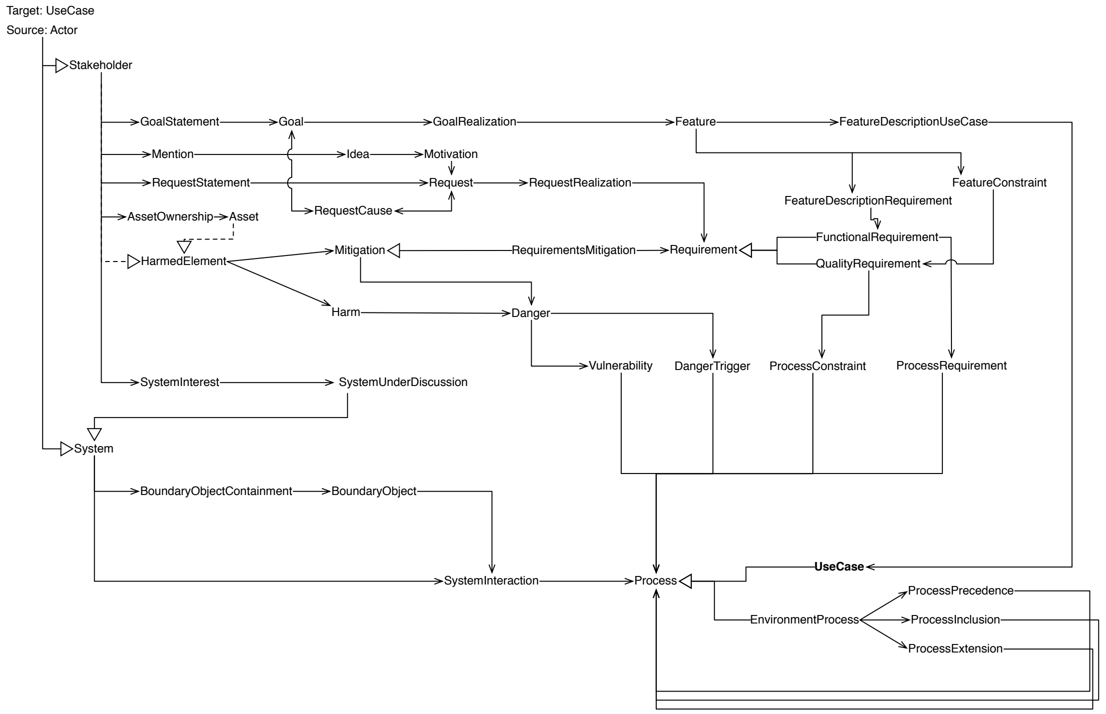


Figure 5.5.: The abstract paths starting from Actor, leading to UseCase.

5.2. Model Quality Evaluation

On a given URML model, a set of quality checks can be executed. A quality check tests a certain rule on the model, and if the rule is violated, the user of the URML tool can be warned. The results of all quality checks give an impression on potential problems of the model. Like in static code analysis, not every violation of a rule is a severe problem, but if numerous violations exist, it may be beneficial to refactor parts of the model.

The configuration of a rule-set for model quality evaluation needs to be project- and process-specific. The requirements analyst has to decide which checks shall be performed, and how the results of each check are interpreted and weighted. The configuration can change over the course of a project. A detailed discussion of the configuration problem is beyond the scope of this dissertation. In general, a model quality evaluation system can be created for any modeling language. Language independent systems can be generalized from the experience with language-specific systems.

In this dissertation, we are interested in evaluation rules that can support elicitation strategies. A metric can support an elicitation strategy or tactic if the strategy or tactic contains rules of the form: If metric M evaluates to value X, proceed with Y, with Y being the next tactic, if the metric is used for guiding a strategy, or the next operation, if the metric is used for guiding a tactic. The abstract rules described below can be concretized for any modeling language but are illustrated with examples based on URML.

5.2.1. Not instantiated entities

This kind of rule can be described by one abstract rule: Does the model not instantiate a certain entity meta-class at all? The abstract rule can be instantiated for any non-abstract entity meta-class. Depending on the type of meta-class, the violation of a rule can say different things about the quality of a model. For example, it is widely accepted that the identification of `Actors` is important. For other kinds of concepts however there is only consensus that they should be modeled depending on the kind of project RE is done for. In the following paragraphs, we discuss a few examples on what a meta-class not being instantiated in a model can mean for its quality.

For example, `Hazards` need not be modeled for systems that have no influence on the physical reality or their influence is negligible. Feature trees need not be modeled if there is no strategy to build a product line.

If the model has no actors, the model is missing knowledge about the roles of users of the system under discussion and the roles of technical systems it is interacting with. Without this knowledge it will be hard to design a system that the target population might want to use. Without identifying the roles of users and interacting systems, there will be problems finding the right people to interview. If the model has no assets, we don't know about the values that stakeholders want to be protected (or not put into danger) by the system.

If the model has no boundary objects, we don't know via which interfaces instances of actors shall eventually interact with the system. Without an analysis of potential interfaces, the design space is completely unconstrained. This is very unlikely if systems exist that the SUD shall interact with. They usually require the usage of existing interfaces (e.g. a certain API that a software system exposes) or require adapters to connect to a legacy interface. Human interface design is constrained by the technical capabilities of the envisioned target platform (e.g. touch displays were unusual 20 years ago, now they are the norm on mobile devices). Apart from the technical capabilities, it is important to understand which role needs which information, and which users might share a certain interface. For example, there might be privacy requirements that constrain use cases and in effect lead to different boundary objects in the model.

5.2.2. Not instantiated relationships

The extreme case of not instantiated relationships is a completely unconnected element. Such an element should either be deleted, or related to an already existing element, or a new element should be created together with a new relationship connecting the two. An element participating in certain relationships can still be "orphaned" with respect to other relationships. Basically orphanhood can be checked for every relationship type. In some cases, orphanhood is not strictly negative: In the case of relationships involving danger, orphanhood could also have a positive connotation, under the strict condition that danger analysis is considered to be complete. A prerequisite for any such check is that there are some entities present in the model that could participate in the relationship in question.

In the following paragraphs, we discuss as examples what non-instantiation of certain relationships means for the quality of an URML model. If instances of Stakeholder (or subclasses) and Asset are present in the model but none of them are related via AssetOwnership, the model is either incomplete or incorrect. It is incomplete, because without knowing which Stakeholder *owns* which Assets, the model is missing information in several aspects: Dangers that *harm* certain Assets negatively affect the Stakeholders that *own* them. For example, if an instance of Threat is saying that access to certain identity information could be exposed in an information system (i.e. the Threat harms an Asset of AssetType identity), the modeler is certainly interested in which Stakeholder this affects and if it is the same Asset that is threatened for every Stakeholder instance. Another example: If there are two Hazards and one *harms* an Asset and the other a Stakeholder directly, mitigation of the the Hazard that *harms* the Stakeholder (i.e. a person or a group of persons) will have higher priority. Furthermore, for any Hazard that *harms* Assets of AssetType property, we can ask whether the Hazard also *harms* the Stakeholder. If we have found an asset but find no matching Stakeholder, this could mean that the Stakeholder has not been identified yet.

If, after these analyses, there are still orphaned Assets, the model is incorrect, because if the model contains Assets that cannot be connected to any Stakeholder in any sensible way, then it is questionable whether the Asset should be in the model at all: If something is *harmed*, but we are can safely ignore whom this affects, we do not have to model the potential harm to the Asset. Such a decision should be done with great care, though. It could be more beneficial to leave the Asset in question in the model, but mark it as out of scope. Orphaned Stakeholders with regard to AssetOwnership can be acceptable if none of their assets is within the scope of the requirements model.

If instances of System (or subclasses) and BoundaryObject are present in the model but none of them are related via BoundaryObjectContainment, the model is rather incomplete than incorrect. Especially the SystemUnderDiscussion should *contain* BoundaryObjects, otherwise there will be no possibility to *interact with* it. For Actors, especially if they model the role of a person, it is unusual to model BoundaryObjects (e.g. eyes, ears), but for special Actors it can make much sense. For a System that shall achieve high accessibility, it is very important to know if its users are in some way handicapped. For `Actors` modeling the role of a technical system, it could also be of interest to model `BoundaryObjects` of the foreign system, in order to understand the interaction with it. E.g. when interacting with a technical `Actor`, we can ask whether the `SUD` ***interacts with*** the other `Actor's` `BoundaryObjects`, or if the `Actor` ***interacts with*** the `SUD's` `BoundaryObjects`, or both. Such questions may also arise when discussing `ServiceProviders`. The available interface of a potential `ServiceProvider` can influence the decision whether it can and shall provide a service to the `SUD` or not. `BoundaryObjects` should always participate in `BoundaryObjectContainment` relationships, otherwise they are a source of ambiguity, as it is not clear which system these objects are an interface to. This is easy to fix if the modeler only forgot to make the connection. But it is also possible that the `BoundaryObject's` existence was forgotten. In that case the missing `BoundaryObjectContainment` can also lead to the identification of missing `Actors` or `ServiceProviders`, and, indirectly, missing `UseCases`. Potentially the missing `BoundaryObjectContainment` means that the unconnected `BoundaryObjects` are duplicates of others, or are modeling relics that can safely be deleted from the model.

If there is no triggering `Process` for a given `Danger`, or a `Process` that does not ***trigger*** any `Danger`, this is not necessarily a model defect. The `Danger` could still be connected to the `Process` with a `Vulnerability` relationship. If however, a `Process` is neither connected via `Vulnerability` nor via `DangerTrigger` to any danger, it has to be reassured if the `Process` is "safe" or whether danger analysis was not done for the `Process`.

There may be `Features` in the model that are not connected to any `QualityRequirement` via the `FeatureConstraint` relationship. But on the opposite, every `QualityRequirement` should be connected to a `Feature`. This can lead to the identification of a new `Feature` that was not modeled yet. The same rule applies to `Feature`, `FunctionalRequirement`, and `FeatureDescriptionRequirement`.

If for a certain `Stakeholder` the model does not contain `Goals` that are linked via `GoalStatement`, it can't be analyzed whether his intentions are supported by the system or whether these not yet modeled `Goals` might even be in conflict with the `Goals` of other `Stakeholders`.

If a `Danger` is not connected to any element via `Harm`, the model is imprecise with regards to who or what is affected by the danger. This can lead to a refinement of the model as new Stakeholders or Assets may be introduced.

5.2.3. Critical Paths

On path completeness checks, we can validate if certain pre-defined critical paths are existing for all elements of a certain kind. This is different from orphanhood, as a defined path can be longer than two entity meta-classes connected by a relationship meta-class.

In URML models, it is for example of interest whether there are `Goals` in the model that have not been operationalized to a `Requirement`. This means that a link in the relationship chain `Goal-Feature-Requirement` does not exist, i.e. either no feature is related to a goal, or a related feature has no related requirements, or both.

Another example from URML would be huge feature models in which isolated feature groups exist that are effectively not rooted in a feature group marked as root. If there were multiple roots in the model, it would be unknown to which of them the isolate feature group belongs.
A more complex example of a path completeness check is related to safety. For each `Danger` related to a `HarmedElement`, we can check whether a `Mitigation` relationship with non-null ends exists, where the mitigated `Danger` equals the harming `Danger`.

5.2.4. Model metrics

On a given model, a set of pre-defined metrics can be continuously be computed. The interpretation of the numbers is up to the requirements analyst, but can also provide input to an elicitation strategy that the elicitation should switch to different topics.

For URML, the percentage of `Goals` being realized in `Features` can be computed. This metric can provide a guideline whether more features need to be elicited. Similarly, the percentage of `Dangers` mitigated by `Requirements` can be computed. This allows analyzing how “dangerous” the currently envisioned system still is.
The percentage of `UseCases` that are related to `Danger` but are mitigated procedurally could motivate asking questions about previously unthought-of mitigations via `FunctionalRequirements`.
The ratio of `Stakeholders` to `Actors` in the model can be analyzed. If the `Requests` of too many `Stakeholders` that will not be using the future system are in the model, the requirements knowledge about how it will really be used may be distorted by the non-user viewpoints. The ratio of `UseCases` to `Features` may indicate that the use case model is too fine-grained.

5.3. Definition of Elicitation Strategies

An elicitation strategy is a plan that describes how to perform a model-driven requirements elicitation. With a strategy, it should be defined what the primary goals are, and how they shall be pursued by means of tactics. A strategy should be flexible enough to deal with the continuously evolving requirements model. For example, when directly talking to stakeholders, there is not always a point in insisting to talk about a certain topic first. If an analyst would for insist on talking only about use cases, when the stakeholder would prefer to talk about goals first, this could lead to resentments that disturb the further proceedings of the elicitation process. An elicitation strategy to react to that situation would be considered more pragmatic than one rigorously enforcing a certain order. In contrast the more rigorous strategy may have the benefit of gathering more knowledge in a certain knowledge area.
Beyond strategies being flexible, the requirements analyst could also switch between strategies, in an ad-hoc manner, at defined milestones, at every elicitation session, or depending on the stakeholder being interviewed.
The strategies outlined below are language-independent, they can be performed with

any language that supports the framework. A language supports the framework by providing information needed to translate elements of the meta-model and the requirements model into natural language questions.

Brute Force Attack For each abstraction of the abstract syntax meta-model, the requirements analyst asks stakeholders regarding instances they know of (i.e. use the *Collect Entities* tactic). A maximum number of instances to be collected for each meta-class should be defined before. The number of instances to be collected for each class can be influenced by the time available. In addition, an upfront prioritization of the abstractions will provide an ordering of the questions.

The up-front prioritization may individually be changed by the analyst depending on his personal impression. For example he might be currently more interested in use cases than stakeholders. As soon as the analyst has collected a number of instances of each concept, *Direct Neighbors* is used to explore the context of each gathered model element. Towards the end of the elicitation process, the analyst uses the framework to compute *Not instantiated relationships* and confirms with each stakeholder that these need not be modeled.

Package-Wise Elicitation This generic strategy assumes that the abstract syntax meta-model of the language is partitioned into packages. Each elicitation session may only be dedicated to the elements of one package. A session can start using the *Collect Entities* tactic, or *Missing Entities First* if some instances already exist for the meta-classes of the package, but only few. If the model is already fairly populated, a good first tactic is *Requirements Analogy* instead. All the tactics are constrained to consider only entities and relationships defined in the chosen package.
As soon as the requirements analyst decides that there is enough information regarding the kind of knowledge corresponding to the meta-model package, the following sessions can target the next package on the list.

Package-Focused Elicitation As Package-Wise Elicitation, this strategy assumes that the abstract syntax meta-model of the language is partitioned into packages. The requirements analyst decides upon one package to be in the focus of attention (called focus package in the following). At the beginning of the strategy, he can begin as in Package-Wise Elicitation, with *Collect Entities*, *Missing Instances First*, or *Direct Neighbors*, depending on the current state of the requirements model. Afterwards, instead of proceeding to another package, the requirements analyst has to decide upon 1-2 central abstractions of each of the other packages, and for each of the entity meta-classes of the focus package, perform the *Source-Target* tactic.

Element-Focused Elicitation The requirements analyst decides upon a meta-class upon which the whole elicitation session will be based. *CollectEntities* will be used only to enumerate instances of the focus meta-class. Then, *Self-Relationships* will be performed if applicable. Afterwards, *Direct Neighbors* explores the context of each of the instances of the focus-meta-class. On any instance found during that *Direct Neighbors* runs, perform *Direct Neighbors* again.

6. Reference Implementation

In this chapter we describe a reference implementation of URML. It provides a pragmatic semantics (see e.g. (Kleppe 2008, 135) for a discussion of the different kinds of semantics) of URML. It is based on a commercial available CASE (Computer Aided Software Engineering) tool and UML. Through its extension mechanism called UML Profiles, UML supports the definition of domain specific languages (Selic 2007). The CASE tool used, Enterprise Architect (Sparx 2014), supports UML and provides an extension mechanism based on UML Profiles.

In section 6.1, we describe how the Enterprise Architect can be extended to support new modeling languages. As we will learn in this section that this extension is largely based on UML profiles, the UML profiles mechanism will be explained in the following subsection (6.2). As Profiles extend UML, any profile creator has to decide exactly which parts are to be extended. Therefore, we provide an overview of UML in section A.8. The discussion which parts of UML are actually good candidates for being extended by URML concepts will be given in section 6.3. Then, in a simplified elicitation process, we describe the environment of our reference implementation, the EA plug-in (6.4). The requirements for the plug-in, its design and implementation are described in the subsequent sections.

6.1. Enterprise Architect and SDK

This section shortly introduces the Enterprise Architect CASE tool (henceforth called EA) and focuses on the description of the extension mechanism that allows the incorporation of a UML Profile-based modeling language to the tool. In the following sections, we call the role of somebody who wants to extend EA the extender. The constituents of a model, the instances of the meta-classes of a language's meta-model, are called simply elements.

Subsection 6.1.1 provides an overview of the concepts that are needed to understand which parts of EA can be extended. How EA's extension mechanism works is explained in subsection 6.1.2.

6.1.1. EA Overview

The main components of EA that the user interacts with are the menu and toolbar items, the toolbox, the diagram editor, and the project browser. Figure 6.1 shows a typical arrangement of these parts in a window, which can be customized. Additional views can be displayed and be hidden and made available for quick access via tabs on

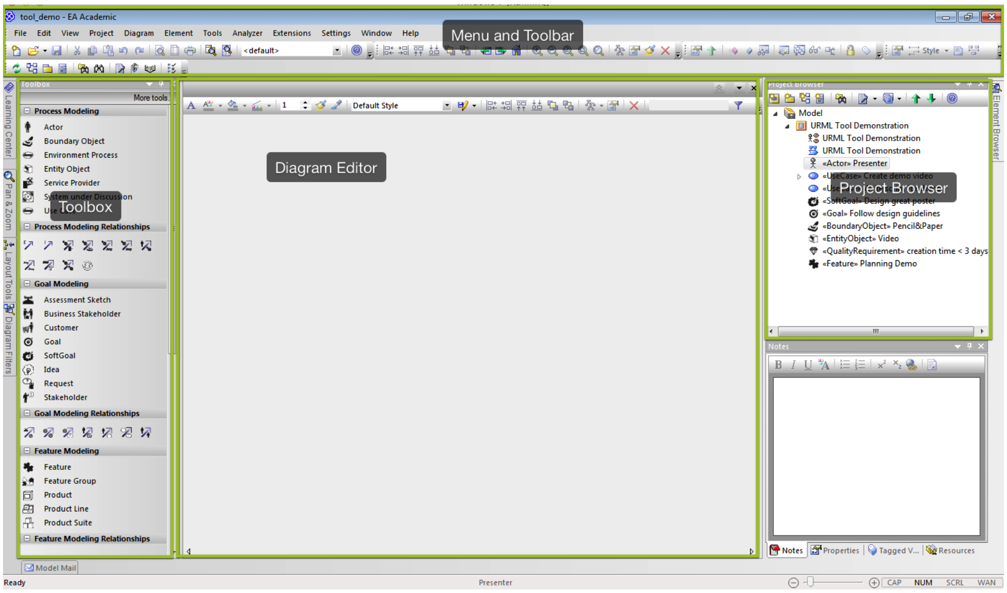
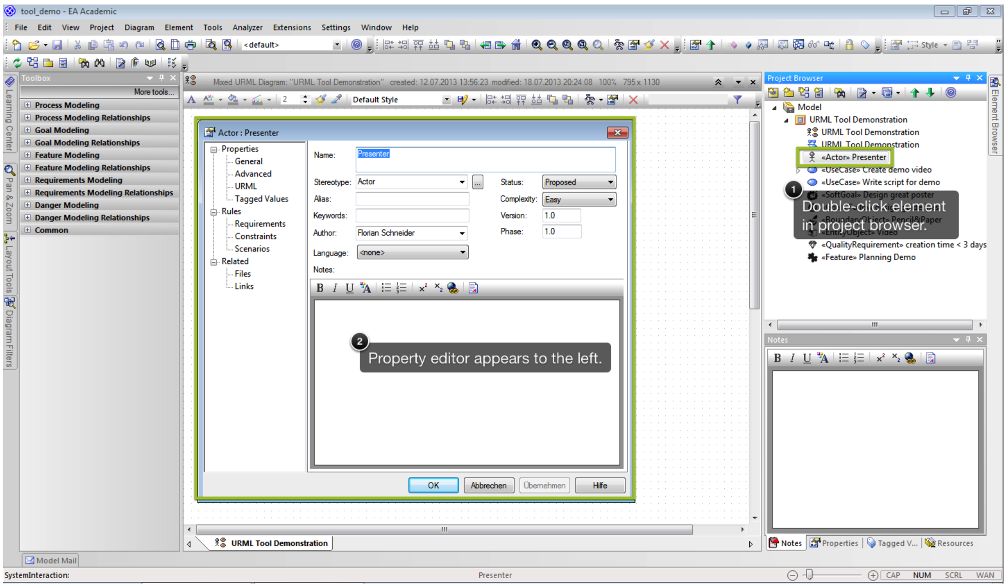


Figure 6.1.: The default layout of the EA main window.

the sidelines of the window. The project browser lists the parts of the project that can be hierarchically organized. Among these parts are (potentially multiple) models and their elements, diagrams, and folders that group any of these. The toolbox enables the creation of entities and relationships on diagrams. Therefore, every meta-class of the language that can be instantiated on a diagram needs a representative in the toolbox, a tool. The tool is displayed as an icon with a tooltip. For compactness of the toolbox, a tool icon is smaller than the graphical representation of an element on a diagram. The tooltip helps the user when she has difficulties to tell from the icon which meta-class of the language is represented by the tool. The toolbox can be thought of as a flattened representation of a language's meta-model. It does allows a grouping of tools, but a tool group has no specific semantics. The toolbox items can be organized in groups, which can be folded and unfolded (see Figure 6.2). These groups are also called toolbox pages.

Double-clicking on other elements of the model opens their property dialog (Fig. 6.2). It shows properties of the element as defined by the modeling language (e.g. Name, Stereotype), but also some EA specific additions (e.g. Status, Complexity). This property dialog is also automatically shown after the creation of a new element, which can be done via the project browser toolbar (the fourth icon from the left) or the diagram editor (described in next paragraph). Double-clicking on a diagram in the browser opens the diagram, i.e. it is displayed and editable in the diagram editor (Fig. 6.3). Existing elements can be added to the open diagram by dragging them from the project browser. New elements can be created by dragging them from the toolbox onto the diagram. Apart from the creation of existing and the addition of new elements, the diagram editor enables lay-outing the diagram. Elements can be moved around the diagram canvas,



Name: Presenter
Stereotype: Actor
Status: Proposed
Alias:
Complexity: Easy
Keywords:
Version: 1.0
Author: Florian Schneider
Phase: 1.0
Language: <none>
Notes:

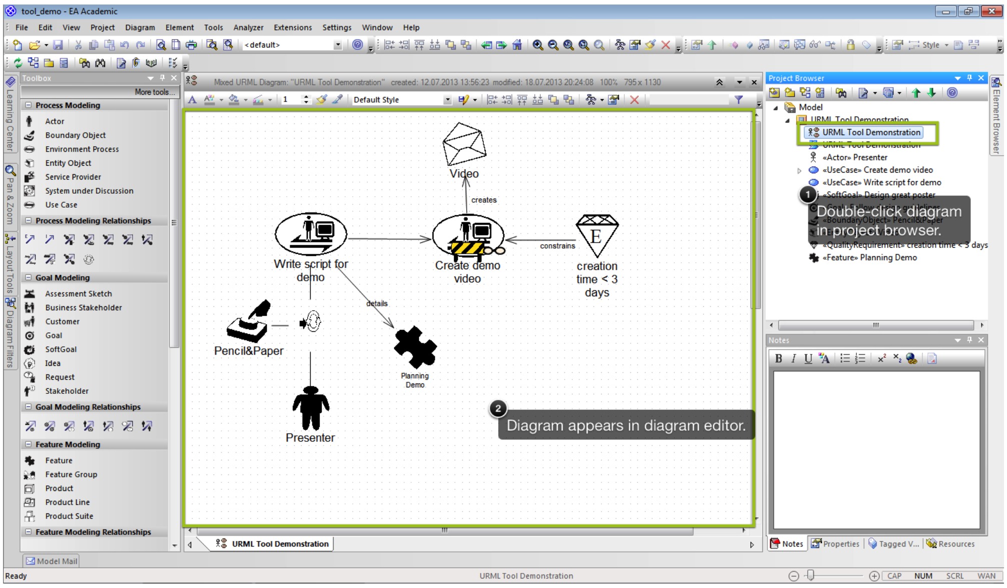
1 Double-click element in project browser.

2 Property editor appears to the left.

tool_demo - EA Academic
File Edit View Project Diagram Element Tools Analyzer Extensions Settings Window Help
Toolbox
Process Modeling
Process Modeling Relationships
Goal Modeling
Goal Modeling Relationships
Feature Modeling
Feature Modeling Relationships
Requirements Modeling
Requirements Modeling Relationships
Danger Modeling
Danger Modeling Relationships
Common
Actor : Presenter
Properties
General
Advanced
URML
Tagged Values
Rules
Requirements
Constraints
Scenarios
Related
Files
Links
OK
Abbrechen
Übernehmen
Hilfe
Project Browser
Model
URML Tool Demonstration
«Actor» Presenter
«UseCase» Create demo video
«UseCase» Write script for demo
«SoftGoal» Design great poster
«QualityRequirement» creation time < 3 days
«Feature» Planning Demo
Notes
URML Tool Demonstration

Figure 6.2.: Property Editor

and any relationships move accordingly. If relationships intersect, the user may define a custom path or try a different auto-layout option offered by EA.



1 Double-click diagram in project browser.

2 Diagram appears in diagram editor.

Figure 6.3.: Diagram Editor

In addition, the diagram editor provides the quick linker, which allows to create new relationships and also new elements, if the relationship shall not connect to an existing element. The quick linker function is evoked from a contextual item that appears when an element is selected in a diagram (Fig. 6.4). Its symbol is an arrow pointing upward. After clicking on the quick linker symbol, the user can drag a line to any place in the diagram (Fig. 6.5).

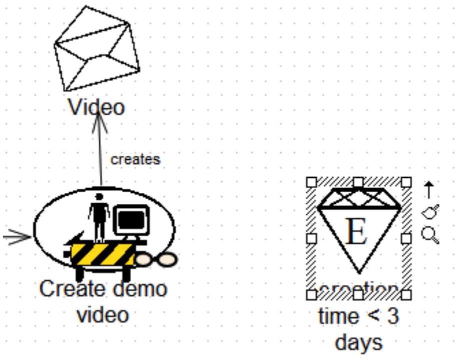


Figure 6.4.: Contextual items displayed on element selection.

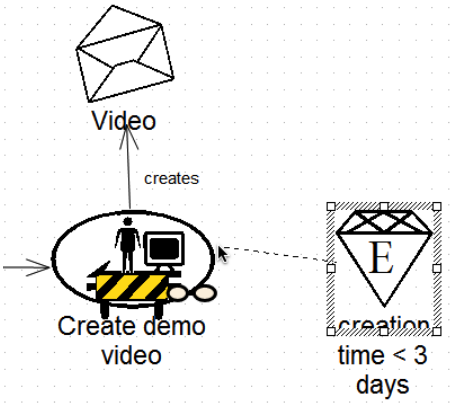


Figure 6.5.: Dragging from the first contextual item, the quick linker.

If she stops dragging above an existing element, the quick linker offers a contextual menu with relationships that can legally be used to connect the two elements (Fig. 6.6). If she stops dragging above an empty area of the diagram, the quick linker offers classes to which the element may legally connect (Fig. 6.7). An instance of the chosen class will be created and placed on the diagram (at the location the user stopped dragging).

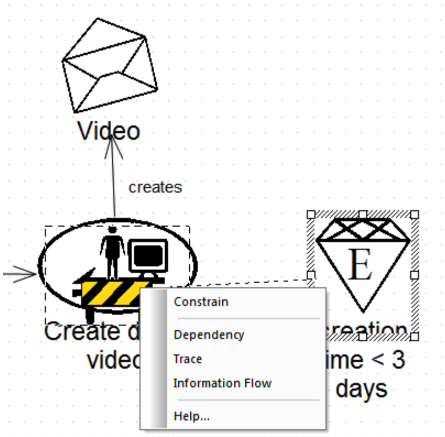


Figure 6.6.: Stopping the drag above another element on the diagram.

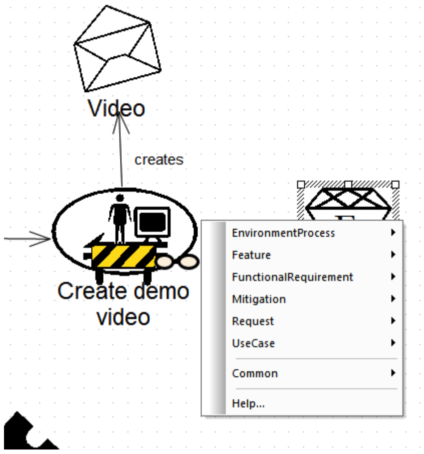


Figure 6.7.: Stopping the drag above an empty diagram area.

6.1.2. EA Extension Mechanism

To add support for a new graphical modeling language to EA, the extender has to create a Model Driven Generation (MDG) Technology. This technology will allow to create diagrams in the language. For this purpose, the extender has to specify a UML profile, a toolbox profile, a diagram profile, and a quick linker definition (Fig. 6.8). This will allow users of an extended EA instance to work with elements and diagrams of the new language. In particular, the MDG technology will specify the elements of the language and their properties, define a group of toolbox pages, a set of diagram types, and some diagram editing behavior.

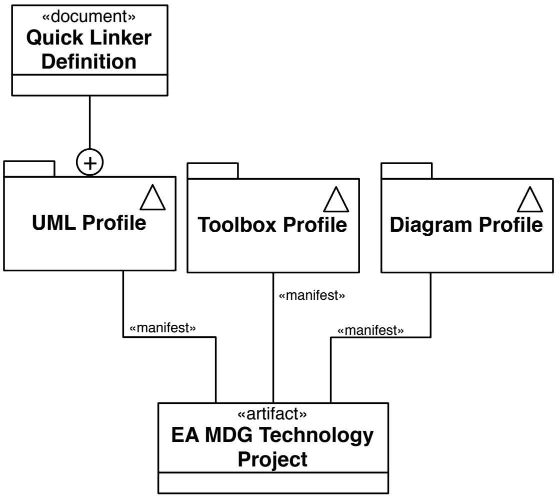
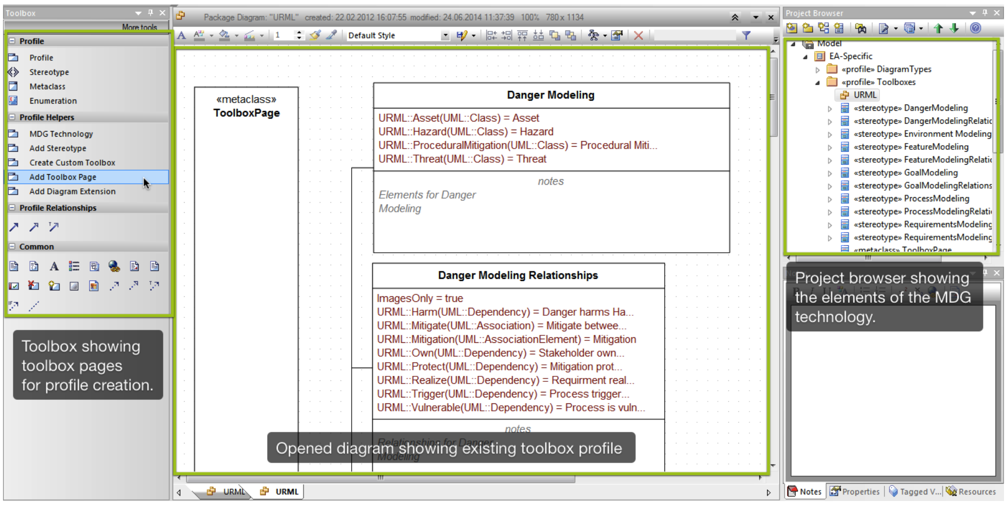


Figure 6.8.: The parts of an EA MDG technology project

An MDG technology is created with EA itself and then deployed to an XML file. To install the technology, the XML file can be placed in a dedicated folder of an existing EA instance. After startup, the EA instance picks up the contents of that folder to provide additional modeling languages to the tool user. Any language specific UI elements have to be coded in a higher level programming language. For this purpose, EA offers an SDK. It can be used to create custom menus, model creation wizards, or tools to check model validity. Such an extension to EA that not only contains an MDG technology, but also language specific UI extensions, is called an Add-In.

To create the MDG technology, the extender creates a project for his extension in EA, and adds one diagram per profile. Dedicated helper dialogs (called profile helpers) support with these diagrams. In particular, they support the tasks of defining stereotypes, creating new diagram types, and creating new toolbox pages. The *UML profile* constitutes the new modeling language as an extension to UML. It defines the elements of the language, their properties, and their appearance on diagrams. The concept of UML profiles is explained in section 6.2. EA adds some EA-specific properties to that mechanism. As defined by UML, a stereotype can be linked to a serialized representation of an image. EA allows this only via an EA-specific mechanism called Shape Script. The script controls the visual appearance of an element on diagrams. It can for example define where the name of the element is placed in relation to its graphical parts. For relationships, EA allows to dictate a certain line style. The line style defines how a line depicting a relationship will be layouted. For example only allowing straight lines for a certain relationship would be enforced by adding the attribute `_lineStyle = direct` to the stereotype. Last, EA allows to add rules to the UML profile to define which connections can be made between elements via the quick linker. The quick linker definition is defined as part of the UML profile, by adding a document to the profile diagram. This document must be named QuickLink in order to work. It must also follow a specific scheme, its format is comma-separated values (CSV). The tools that the language adds to the toolbox are specified in a *toolbox profile*. The toolbox profile consists of a set of toolbox pages. For the visual representation of a toolbox page, the extender can specify a name, a tooltip, an icon, and some display options. Every toolbox page consists of a set of tools. For the visual representation of a tool, the extender can specify an alias and a tooltip. The alias is a a short name that is displayed instead of the fully qualified name. For the determination of which stereotype gets instantiated when the user selects the tool, the extender has to provide a reference to an element of the UML profile. An exemplary workflow to create a toolbox page with one tool is highlighted by Figures 6.9, 6.10, and 6.11.



Package Diagram: "URML" created: 22.02.2012 16:07:55 modified: 24.06.2014 11:37:39 100% 780 x 1134

«metaclass»
ToolboxPage

Danger Modeling

URML::Asset(UML::Class) = Asset
URML::Hazard(UML::Class) = Hazard
URML::ProceduralMitigation(UML::Class) = Procedural Miti...
URML::Threat(UML::Class) = Threat

notes
Elements for Danger Modeling

Danger Modeling Relationships

ImagesOnly = true
URML::Harm(UML::Dependency) = Danger harms Ha...
URML::Mitigate(UML::Association) = Mitigate betwee...
URML::Mitigation(UML::AssociationElement) = Mitigation
URML::Own(UML::Dependency) = Stakeholder own...
URML::Protect(UML::Dependency) = Mitigation prot...
URML::Realize(UML::Dependency) = Requirment real...
URML::Trigger(UML::Dependency) = Process trigger...
URML::Vulnerable(UML::Dependency) = Process is vuln...

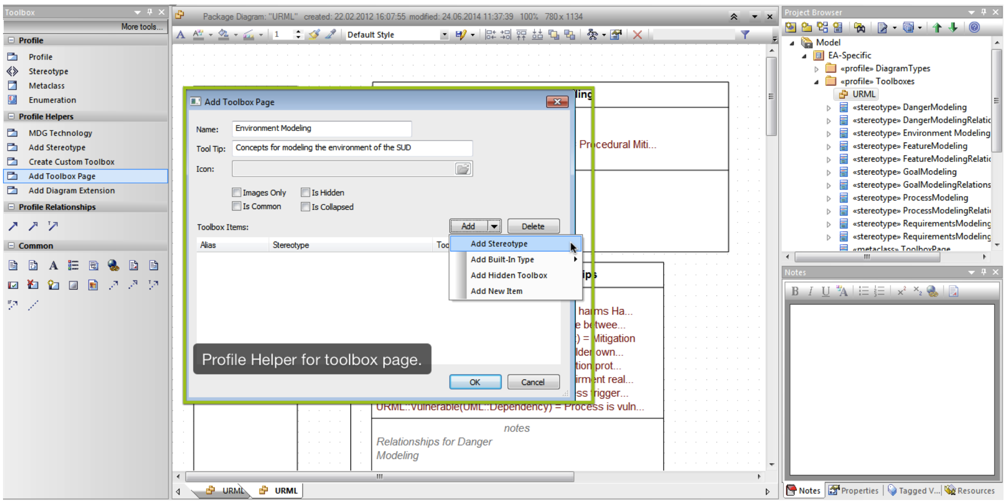
notes
Relationships for Danger Modeling

Toolbox showing toolbox pages for profile creation.

Opened diagram showing existing toolbox profile

Project browser showing the elements of the MDG technology.

Figure 6.9.: Editing the toolbox profile



Package Diagram: "URML" created: 22.02.2012 16:07:55 modified: 24.06.2014 11:37:39 100% 780 x 1134

Add Toolbox Page

Name: Environment Modeling

Tool Tip: Concepts for modeling the environment of the SUD

Icon:

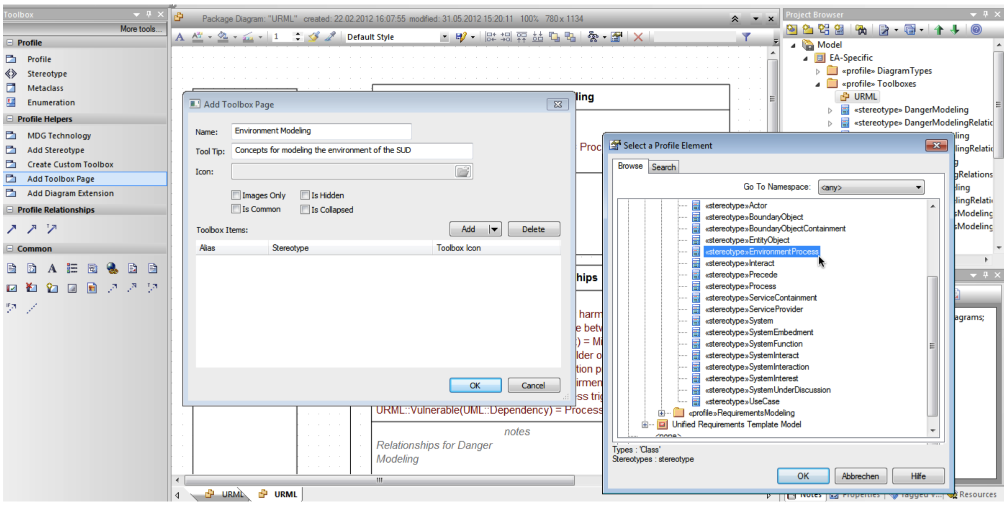
□ Images Only □ Is Hidden
□ Is Common □ Is Collapsed

Toolbox Items:

Alias	Stereotype	Too

Profile Helper for toolbox page.

Figure 6.10.: Profile helper for toolbox pages



Name: Environment Modeling

Tool Tip: Concepts for modeling the environment of the SUD

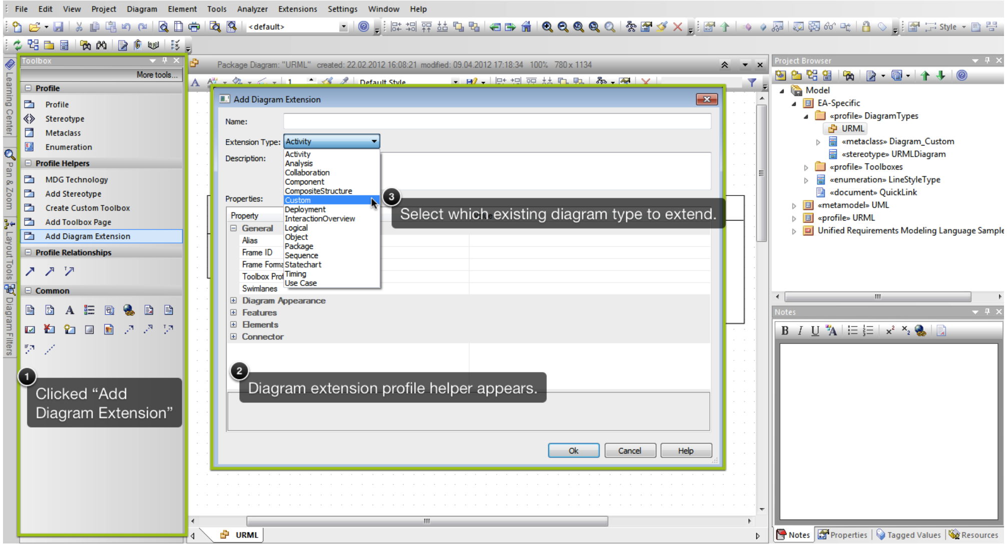
Icon:

Go To Namespace: <any>

Types : 'Class'
Stereotypes : stereotype

Figure 6.11.: Selecting a stereotype to be presented in a toolbox page

The *diagram profile* specifies the kinds of diagrams the extension allows to create. The extender can choose to extend existing diagram types offered by EA, or to create a custom diagram type (Fig. 6.12). Diagram types are displayed in dialogs with which the tool user will create new diagrams. A diagram profile however does not restrict the user what to put on a diagram. It only suggests what should be put on the diagram: a diagram profile is bound to a toolbox page, which makes the bound toolbox page appear in the toolbox when the diagram is created. Users can still select a different toolbox page, so the binding is not strict.



Extension Type: Activity

1 Clicked “Add Diagram Extension”

2 Diagram extension profile helper appears.

3 Select which existing diagram type to extend.

Figure 6.12.: Selecting a stereotype to be presented in a toolbox page

6.2. The UML Profile Mechanism

This section provides an overview of on UML and UML profiles, an extension mechanism of UML 2.4.1. Though the mechanism is in theory available to any language based on MOF, it currently is defined in the specification documents of UML (OMG 2011c; 2011b). We therefore explain profiles in terms of UML. This section represents our interpretation of the UML superstructure specification (OMG 2011c, Fig.12.2). In our figures, we follow the convention that meta-classes are annotated by a ≪class≫ keyword and meta-associations are annotated by a ≪association≫ keyword and have a different background color. To indicate the relationships of elements of the Profiles package to elements of other UML packages, those elements are presented in grey color. In figures with a focus on inheritance hierarchies, classes that are leaves of the inheritance hierarchy have a bold outline.

A `Profile` is a special kind of `Package` that refers to a reference meta-model in terms of `PackageImport` and/or `ElementImport` relationships (Fig. 6.13). Via these relationships, the profile imports elements from other UML packages. These imported elements can be extended by the `Stereotypes` of the profile. Second, the profile establishes filtering rules, which will be explained in a later paragraph.

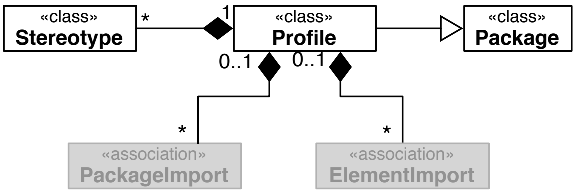


Figure 6.13.: Basic structure of the profile mechanism.

A stereotype is a special kind of `Class`, that extends other classes not by inheritance, but by `Extension` (Fig. 6.14). This construction is needed to differentiate between the original UML model and its extensions. Instances of stereotypes cannot exist without an instance of the extended meta-class. Conversely, instances of the extended meta-class may exist without the extending stereotype. If the extension is not required as indicated by the `isRequired` attribute, the modeler may choose whether the CASE tool should instantiate the stereotype or not. Extension is binary and can only occur between a class and a stereotype, not between classes and also not between stereotypes. This is why the `ExtensionEnd` meta-class is needed, which is explained in a later paragraph.

It is important to note that when talking about profiles, we often talk about the effects of profile application. In the case of stereotypes, we talk about how an instance of a stereotype alters an instance of a class. A stereotype can alter the extended class by adding additional properties and/or altering the class' visual appearance. As a stereotype can have properties (it inherits from `Class`), the class instance obtains new attributes or associations when the stereotype instance is added. In addition, a stereotype can specify `Images` that alter the graphical representation of instances of the extended meta-class, or even completely replace the standard notation. How and under which circumstances the images are applied, is left as an implementation decision to tool vendors. Through the inheritance from class, stereotypes may also use inheritance, but only to inherit from other stereotypes. Furthermore, as `Stereotype` is a subclass of `Class`, it can have properties typed by classes defined in or imported by the profile.

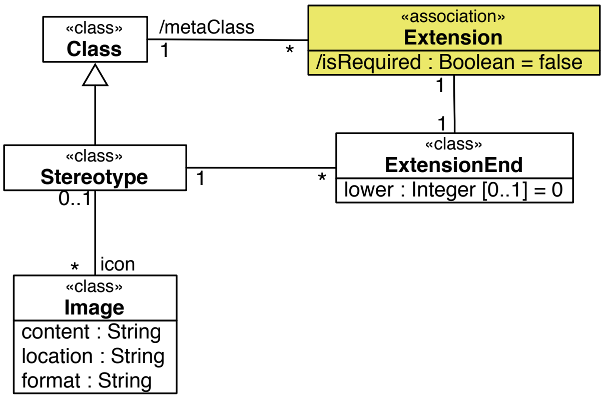


Figure 6.14.: Stereotypes as extensions to classes.

`Extension` is a special kind of `Association` (Fig. 6.15). It specializes `Association` by placing a constraint on the number of `memberEnds`: it has exactly two (i.e. it is binary). Furthermore it redefines `ownedEnds`. Through the redefinition, it reduces the number of `ownedEnds` to one. That one end must be an `ExtensionEnd`. `ExtensionEnd` is a special kind of `Property`, which can only have zero or one as lower multiplicity bound. The `isRequired` attribute of `Extension` is derived from that lower multiplicity bound: If it is 0 the extension is optional, or required if it equals 1. As previously noted, tools then decide on the basis of `isRequired`, whether users have the choice to apply a stereotype or whether the stereotype is automatically applied.

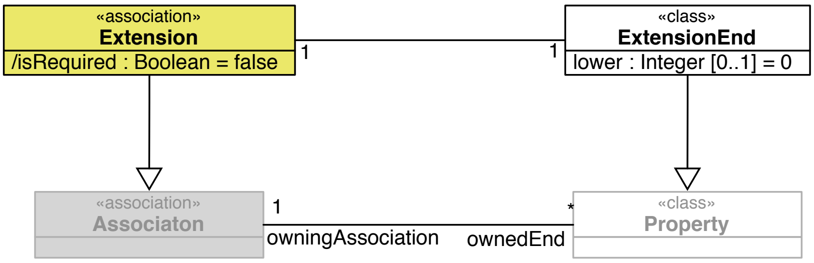


Figure 6.15.: A detailed look at Extension.

Whenever a CASE tool user decides to use a profile in addition to the UML, one to many instances of `ProfileApplication` are created (Fig. 6.16). The profile application defines how the package, to which the profile was applied, is altered. Profiles never alter packages destructively. They cannot remove parts of a model, only add new parts or hide certain parts temporarily (i.e. instances of certain classes are not shown as long as the profile is applied). A profile application leads to a potential extension of applicable classes (those that the profile refers to via package or element imports) that may receive new properties then and may appear visually different. An applied profile may hide certain parts from the applying package if `isStrict` is set to true. That hiding is called filtering rules. These rules are based upon the meta-classes and packages imported by the profile and the extensions it defines. A class of a model with a profile applied is only shown on diagrams if it is extended by a stereotype (where `isRequired` = true or the user decided to use the stereotype), or explicitly imported via `ElementImport` by the profile, or indirectly imported by the profile because its containing package was imported. The last rule has an exception: If the profile imports a package via `PackageImport` and some of its contained classes via `ElementImport`, only instances of those explicitly imported are visible on diagrams.

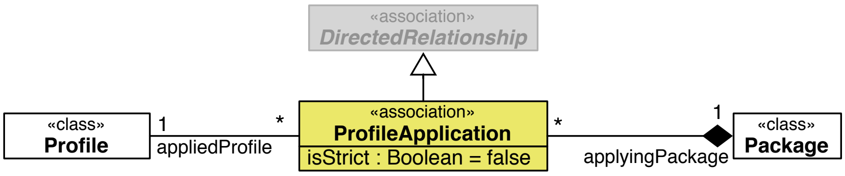


Figure 6.16.: ProfileApplication is a relationship between a package and a profile.

When transforming the URML meta-model to a UML profile, we need to decide for every stereotype, which UML class it shall extend. An extended UML class is called *base class*. By definition of the profile mechanism, a profile should be “conformant with [...] the semantics and abstract syntax of UML” (Selic 2007). Therefore a base class needs to be carefully selected. As UML is a complex language[1], this is a challenging problem. All classes of the UML should at least be considered, each could be a valid base class for a URML stereotype. UML 2.x up to version 2.4.1 was partitioned into compliance levels L0, L1, L2, and L3. As tool implementations should usually claim UML compliance relative to these levels, it is a sensible consideration for a UML profile, on which compliance level to base upon. Using a lower compliance level would simplify the task of choosing base classes, as lower levels have fewer classes. E.g. UML 2.4.1 L0 has only 6 relationship meta-classes as opposed to 25 on level 3 (See Appendix Figures 6.17 and A.146).

[1] In version 2.4.1, at compliance level 3, UML consists of 673 elements. This number results from counting the number of `packagedElement` tags in (OMG 2014b). It can be broken down to 242 classes, 418 associations, and 13 enumerations. These numbers result from inspecting the xmi:type attribute of the aforementioned `packagedElement` tags, with `xmi:type="uml:Association"` or `"uml:Class"` or `"uml:Enumeration"`.

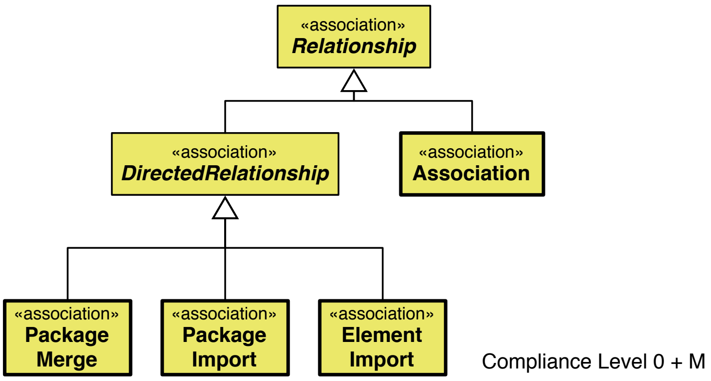


Figure 6.17.: UML relationship meta-classes at compliance Level 0.

To support a quick dissemination of our language, we had the goal of choosing UML base classes such that URML support could be added to any CASE tool. This would motivate choosing the compliance level that most existing CASE tools adopt. However, there is no extensive data on the degree of UML compliance of existing CASE tools. To the best of our knowledge, the most recent study that investigates tool compliance with UML 2.1.1 was done several years ago (Eichelberger, Eldogan, and Schmid 2009). The results of this study suggest that most tools do not achieve full UML compliance, and many not even a basic level of compliance. That may have changed in the meantime, even more so as the compliance levels were eliminated in UML version 2.5 (OMG 2015c). Therefore, the former compliance levels of UML could not be used as a criterion for the selection of base classes for URML stereotypes. All classes of UML had to be considered as potential base classes.

A full discussion of the UML meta-model is beyond the scope of this thesis. To provide some background knowledge for the choices that are presented in section 6.3, an overview on the UML meta-classes is given in Appendix Section A.8.

6.3. Transformation of the URML Meta-Model to a UML Profile

This section describes the mapping of the URML meta-model to a UML profile. The list of potential base classes of UML is narrowed down by excluding UML classes that have no semantic commonality with URML meta-classes. The remaining classes are tersely discussed in terms of their suitability as a base class. A complete mapping specification is given in the appendix A.7.

To narrow down the set of potential base classes, we excluded all meta-classes from the candidate list whose semantics are too software implementation specific to be meaningfully extended by meta-classes with requirements-oriented semantics. Among those are UML concepts abstracting from to programming language concepts like control flow (i.e. activities and interactions), modularization techniques, finite automata, and templates.

For example, we can exclude ProtocolConformance, TemplateBinding, ProfileApplication, ElementImport, PackageImport, and PackageMerge from Figure A.146. With the same arguments, we can exclude Clause, ExceptionHandler, LinkEndData, QualifierValue (used for modeling software control flow), all concepts related to parametric polymorphism (ParameterableElement, TemplateableElement, TemplateParameter, TemplateParameterSubstitution, TemplateSignature), and Slot (which is used for modeling objects at runtime) from Figure A.149.

From the remaining candidate classes, we have to take into account that stereotypes on base classes that are high up in an inheritance hierarchy will have a greater effect than stereotypes on base classes lower in the inheritance hierarchy. Using a base class too high in the hierarchy might have unwanted side effects. We could think of a URML stereotype Requirement having PackageableElement as a base class, as it facilitates the grouping of elements in packages - a characteristic that also elements of requirements models could benefit from. This would make the Requirement stereotype applicable to instances of all subclasses of PackageableElement - so *Requirement* would effectively also extend Event, for example (see Fig. A.160 for all subclasses of PackageableElement). This seems to be the wrong way to think about stereotypes: In the previous argument we argued that *Requirement* should stereotype PackageableElement because it should also be packageable. But because of that one-dimensional argumentation, we only extend the characteristic of packageability that is captured in the abstract class PackageableElement. Instead, we should extend some class that inherits packageability, but is more concrete. Unfortunately, the more concrete a UML meta-class is, the more semantics it inherits, because of UML's deep inheritance hierarchy. This is even more complicated by the fact that UML supports multiple inheritance. So we have to investigate the complete semantics of a meta-class (by taking all its superclasses into account) before discussing its suitability for extension. Only then we can decide whether a meta-class of URML could stereotype it. We show how complicated such an investigation can become by example of the Class meta-class. As we can see in Figure 6.18 it has 12 abstract meta-classes in the inheritance hierarchy above it. Would we make *Requirement* a stereotype of Class, it would immediately become parameterable, templateable, packageable, redefinable, would constitute a namespace, a type, have structure and behavior, and possibly be encapsulated. Because Class is on a very low level of the hierarchy (i.e. it has many super classes) it has very specific semantics. To create a UML profile that constitutes a requirements modeling language, our task is to search for leaves of UML's inheritance hierarchy, that are not too software-design specific yet.

Unfortunately, the leaves of UML's inheritance hierarchy are - with a few exceptions - all intended for the modeling of structure and behavior of software. Thus they contain no suitable base classes except for Actor, UseCase, Include, and Extend. These can be used for the corresponding URML concepts *Actor*, *UseCase*, *ProcessExtension*, and *ProcessInclusion*. For all other concepts of URML, UML has no suitable base class, as the extension would create semantic conflicts. For example, we based *FunctionalRequirement* upon Class. Still we decided to create a URML UML Profile with with the base classes Actor, Association, Class, Component, Dependency, and Information-

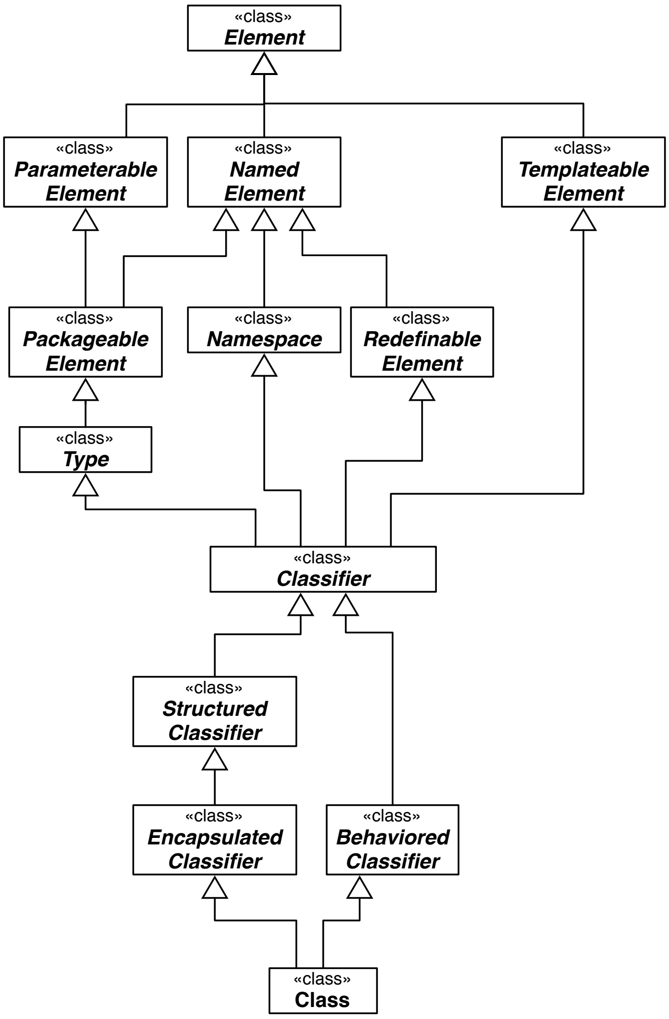


Figure 6.18.: The superclasses of `Class`.

`Flow`. We thus created a lot of semantic conflicts in our reference implementation. This will be discussed in more detail in section 6.5.

6.4. Description and Conceptual Model of the URML Extension to Enterprise Architect

In this section we provide a conceptual model of the URML reference implementation, the URML EA Plug-In (henceforth called *extension*). In the subsections we explain use cases of the *extended tool*. For each use case, we show what parts of the tool (toolbox, project browser, diagram editor, property dialog, quick linker menu, main menu) are affected by the extension. Most of what we describe here is not specific to the URML extension but applies to any extension implemented as MDG technology.

In general, the tool organizes models in projects. Within a project, the user can manage a set of models and can generate documentation from models (Fig. 6.19). The extension does not modify how documentation is generated from models, which is why we do not discuss the use case `Generate Documentation` in this section. When managing a model, the user manages elements, diagrams and the structure of the model. The structure of a model is given by a hierarchical grouping of elements and diagrams in folders, and the ordering of the elements, diagrams, and folders within folders. The user interface component to provide access to the model structure is EA's project browser. The project browser is not affected very much by the plug-in. Only the icons of the elements shown in the project browser are taken from the definition of the UML profile and, for diagrams, from the diagram profile. In all other aspects, the project browser is not modified by the extension. Therefore we do not discuss the use case `Manage Model Structure` in more detail.

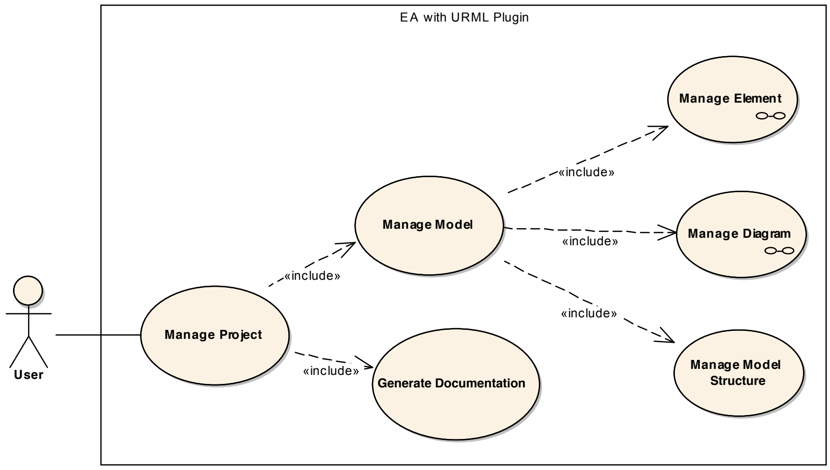


Figure 6.19.: General CASE tool functionality offered by EA.

Regarding the management of diagrams, the user can create, delete, read, and edit diagrams and highlight a given diagram in its enclosing folder in the model structure (Fig. 6.20). `Create Diagram` is extended insofar that diagram types defined by the diagram profile of the extension are shown in the diagram creation wizard of EA. `Delete Diagram` and `Highlight in Folder`[2] are unaffected by the extension. The `Read Diagram` use case is addressed by the diagram editor of EA: It provides a scrollable canvas in which the diagram's elements can be viewed, related and edited, and in which the layout of the diagram can be defined. The latter three functions are explained in the next paragraph that deals with diagrams (Fig. 6.21).

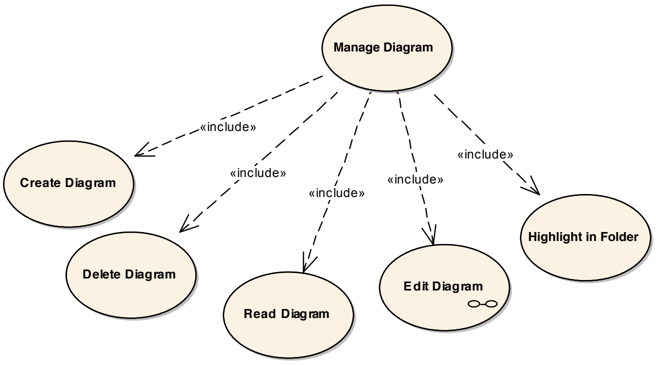


Figure 6.20.: Working with Diagrams.

The structural and behavioral properties of a diagram in EA are not affected by the extension. The addition of an existing model component to a diagram is also unchanged. The addition of new elements is affected by the UML profile, as it defines what entities can be created and how they can be related. The creation of elements will be explained in the next paragraph. Relating elements includes instantiating a relationship, so this aspect will be discussed there as well. We will focus here on a specific behavior offered by the diagram editor: It provides access to the quick linker, which is a contextual menu that allows the creation of relationships from existing elements. The created relationship may create a connection to another existing element, but the quick linker also supports the creation of new elements. Which relationships are offered by the quick linker is defined by the quick linker rules attached to the UML profile. These rules are encoded in a Quick Linker Definition document, to which the profile contains a link (see Fig. 6.8). The layout of diagrams is partially affected by the UML profile, which potentially prescribes the line style of the line that visualizes a relationship.

[2] As the name of the use case is not self-explaining, we give a short explanation here: `Highlight in Folder` provides a navigation path between the diagram editor and the project browser: The function can be chosen from a contextual menu on the diagram, then makes the enclosing folder visible and opened in the project browser, then navigates to the diagram there, and selects it.

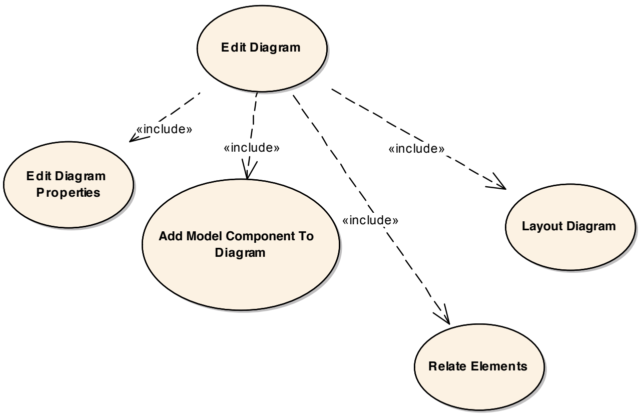


Figure 6.21.: Editing Diagrams.

The use cases for the management of model elements are presented in Figure 6.22. The creation of new elements is governed by the UML profile. It defines what can be created and thus defines what can appear in the element creation dialog of EA and the toolbox. In the creation dialog, URML meta-classes appear only textually. In the toolbox, meta-classes appear visually as tools, which are defined in the toolbox profile of the extension. The editable properties of an element, apart from those already defined by UML are defined by the UML profile. `Delete Element`, `Find Displaying Diagrams`, and `Highlight in Folder` are unchanged by the extension.

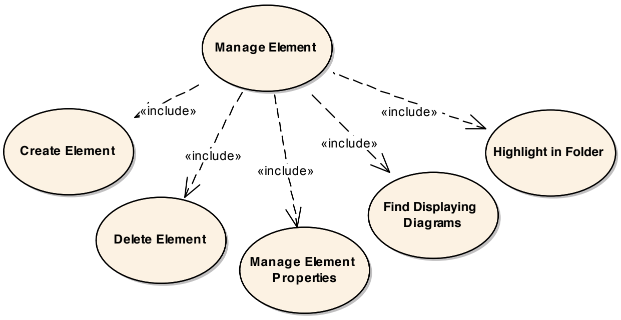


Figure 6.22.: Working with Model Elements.

6.5. Discussion

Our choice of the Enterprise Architect CASE tool as a basis for our reference implementation was entirely driven by pragmatic considerations. We wanted to conduct exploratory studies at Siemens, to test the applicability of URML to real-world problems. This enforced the strict requirement to extend the Enterprise Architect CASE tool, as that was the tool being used in the units we wanted to perform the studies with. We could not conduct studies with pencil and paper as the notation of URML is of low sketchability, i.e. the models cannot be hand-drawn easily. In this thesis we have to leave the question unanswered whether that is an important feature for requirements modeling languages. Tool support will anyhow often be needed - no matter if a language has sketchability or not. Tools can support syntax checks, the collaboration on models, version control, and complexity management. From our experience with extending EA, we derive another interesting question: How much of the tool support can be predefined within the language itself? Certainly a language definition is not able to shape every thinkable use of the language by users. But in order to achieve tool interoperability and platform independence, we can conclude that the more is predefined within the language definition, the easier it will be to achieve these goals. In our case, should the reference implementation be ported to another tool, validation rules, the quick linker, the toolboxes, and diagram definitions would have to be created anew, specifically for each targeted tool. A side note on the toolbox and diagram profiles of EA: Their naming is misleading as it suggests they are also relying upon the UML Profile mechanism. This is not the case. While the mechanism is similar to UML profiling, the base classes available (`ToolboxPage` and `Diagram_Custom`) are specific to EA. The UML profile of an MDG technology is also partially tool-specific: The addition of a class with document stereotype to the profile in order to enable the quick linker mechanism is an EA-specific mechanism.

A higher portability can potentially be achieved by a transformation-driven approach. Such an approach would be capable of generating the files needed to extend various CASE tool from a language meta-model. In such an approach, a new transformation would be written every time a new CASE tool should be supported, and existing transformations would need to be modified for every major version of an already supported CASE tool. Instead of porting the EA specific UML profile to e.g. MagicDraw, we could then use software engineering techniques to reuse as much of the existing transformations as possible.

The choice of EA for the reference implementation also forced us to create a URML UML profile.[3] In hindsight, the suitability of URML as an extension to UML is rather low. It is semantically very different from UML, which should be expectable for a language that models concepts related to requirements analysis. With UML, software systems can be described. The modeler can start with models rather sketchy in nature and proceed to a very exact model that can be translated to source code. But even a sketchy model of a software system has not much in common with a model of that system's requirements. That SysML is realized as a UML profile does only show in our opinion, that OMG, an organization with many tool vendors among its members, is led by considerations where compatibility with the existing tool landscape is more important than a conceptually clean foundation of the language.

But a truly universal modeling language for systems would contain concepts for modeling artifacts of the whole development lifecycle. There are many concepts that can be

[3] We even thought about realizing the URML reference implementation as a profile on SysML instead of on UML, as SysML already contains a concept for requirement and is more oriented towards systems engineering. But we found no CASE tool to support this, so this would not have been a possibility for us even if we weren't bound to do the implementation on the basis of Enterprise Architect.

modeled during the development process but are no specialization of concepts modeling the properties of a system under discussion: For example its requirements, the business process the SUD is being used in, the requirements of the development process, the procedural mitigations of hazards, or business rules and goals. For such concepts it seems awkward to extend meta-classes of UML like `Class` or `Dependency`. Finding the right UML base classes, has also proven to be a hard problem - given the complexity of UML. It is very hard to avoid unwanted side effects. I.e. as many stereotypes of our UML profile extend `Class`, URML entities can be connected in any way with any instance of Class on diagrams that mix UML and URML. For example, a `Class` instance could inherit from a `Goal` instance.

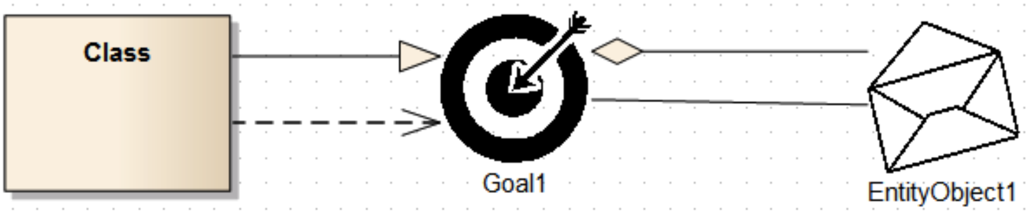


Figure 6.23.: Unwanted side effects

Furthermore, unwanted connections between entities of URML are also possible. For example, an `EntityObject` could be contained in a `Goal`. There is no way to prevent that from happening, without further additions to the profile or custom code that checks validity of links upon creation. Profile additions would be non-standard as UML profiles offer no mechanism for excluding certain stereotypes from certain relationships. This is not a weakness of the profiling mechanism per se. It rather shows that we made unintended use of the mechanism. Or current approach to address the problem (custom code) is embodied by a software module added to the MDG technology, making use of the SDK of Enterprise Architect to avoid unwanted connections at runtime. That software module makes our reference implementation even more tool specific.

We conclude that declaring a domain-specific language on the basis of UML is not a good choice for every language. Only languages that in very strict sense specialize UML should follow this path. Therefore we reason that future work on URML tool support should focus on making the language independent from UML. For instance, an implementation on the basis of MOF (OMG 2013a) or GOPPRR (Kelly 1997) could be investigated. Following these paths however will come at the cost that a future reference implementation of URML will need more software development efforts, as CASE tools as EA do not offer extensibility on the basis of these two meta-languages.

7. Towards an Evaluation of URML

This chapter describes how the URML has been evaluated to date. A comprehensive empirical validation of the language has been beyond the scope of this dissertation. Within its scope, we have undertaken two efforts to evaluate URML: First we have used URML ourselves on realistic examples to evaluate the applicability of URML for expressing requirements knowledge. Of the examples that we have created, three are presented in Section 7.1. One out of the three examples comes closest to a real case study, as a subject matter expert has been involved. The bCMS example problem is notable as it facilitates a model-wise comparison of URML, KAOS and URN. The KAOS and URN models were presented in sections 3.1.6 and 3.2.3. The respective presentation of the bCMS problem in URML is provided in section 7.1.3. Apart from these studies, the URML has been evaluated by a requirements engineering expert, in two requirements models for real medical systems (Berenbach, personal communication, 2012-2013). Unfortunately these models were not published for intellectual property reasons.

As an important goal of this dissertation has been the creation of notational symbols with semantic immediacy, we have conducted an experiment to evaluate the entity symbols of URML. That experiment also provides a comparison to two other requirements modeling languages, KAOS and URN, by conducting the same experiment with these languages as well. The experiment is described in the second part of this chapter, in Section 7.2. A comprehensive evaluation of the learnability of URML will need to test the influence of Moody's other principles, e.g. cognitive fit, complexity management, cognitive integration, and graphic economy. A fair comparison with other languages will also take aspects as the functionality of a language into account.

7.1. Modeling Studies

To illustrate the applicability of URML, we have performed three modeling studies. The first one was inspired by an accident report of the Occupational Health and Safety Administration (OSHA), in which we have reverse engineered the requirements knowledge regarding a sausage machine from the accident report and two manuals on sausage machines. The second study involved gathering requirements knowledge from a real expert in the medical domain. It is about the blood sample management process inside a hospital. Especially this study is an example with a high degree of realism. The third study is an example problem that has been subject of the the CMA@RE workshop (Moreira, Ana, Georg, Geri, and Mussbacher, Gunter 2013). As we have already presented KAOS and URN models of that problem, the respective subsection serves to illustrate a comparable model expressed with URML.

7.1.1. Sausage Stuffing Machine Accident

An accident in the kitchen of a restaurant in December 2011 was inspected by the Occupational Health and Safety Administration (OSHA) and reported as inspection 315773630(OSHA 2011). The accident happened during the cleaning of an electric sausage stuffing machine. That machine consists of the following major parts: The stuffer tray receives the raw material that makes the sausage, a screw conveyor transports the material to the stuffer tube, which holds the sausage casing, the transported raw material is pressed into a sausage casing, a motor turns the screw conveyor. The motor is operated by a foot pedal and is connected to the electricity network via a standard power supply. What happened due to the accident report: The cleaning worker, flushing the stuffer tray with water, saw particles on the screw conveyor and used his right index finger to put them away. He then slipped, as water was on the floor, and accidentally stepped on the foot pedal, which started operation of the machine. The finger had to be amputated.

Regarding the circumstances that led to the accident, the employer was cited for violations of four federal regulations, published in Title 8 of the California Code of Regulations (T8 CCR) (OAL 2014). In particular, “section 3314 (c) for Employer’s failure to ensure that its employee disengaged or deenergized the power source of a sausage machine by unplugging the machine prior to cleaning”, “section 4184 (b) for employer’s failure to provide guarding of the sausage machine at the point of operation”, “section 4185 for employer’s failure to provide a safeguarding device for its sausage machine’s foot-pedal”, “section 3203(a) for Employer’s failure to establish an Injury and Illness Prevention Program for its employees”, and “section 3314 (g) for Employer’s failure to develop hazardous energy control procedures for cleaning, repairing, servicing, and setting up machinery or equipment used” (OSHA 2011).

In the following, we will transform the textual information taken from (OSHA 2011) into a URML model. To support a post mortem analysis of the accident, we reverse engineer parts of a requirements analysis model of a sausage machine, using the information given in the inspection (OSHA 2011) and a manual of a sausage stuffing machine (Talsa 2006)

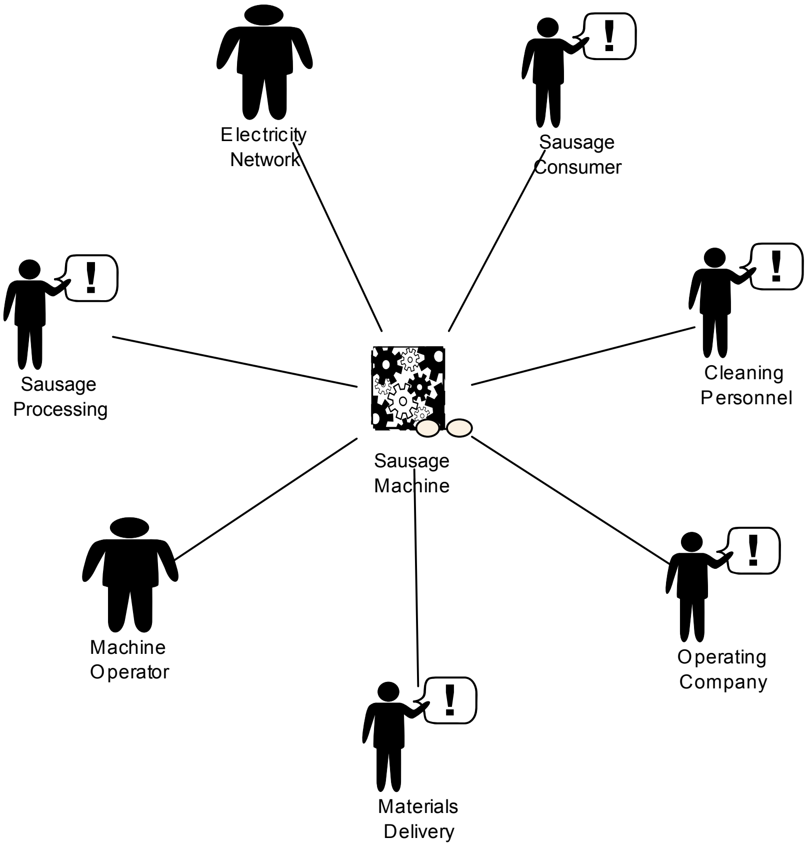


Figure 7.1.: Stakeholders of a sausage stuffing machine

We start with analyzing the context of an electric sausage machine in Figure 7.1. It has two actors: Because it needs electricity it is connected to the `Electricity Network`, with which it interacts to consume power. The second actor is the `Machine Operator`, the role of somebody using the machine to produce sausage. We then consider stakeholders that are not directly interfacing with or using the machine. The `Operating Company` has bought the machine in order to provide a service that is supported by the machine. It employs the `Machine Operator` and also the `Cleaning Personnel`. The `Cleaning Personnel` is no actor of the machine, as it neither uses one of the machine's functions nor provides something that is required by one of its functions. It is an interesting example of how the system boundary affects the model. Would the system under discussion be the process of operating and maintaining the machine, the `Cleaning Personnel` would be a service provider. It is also an example of how the available functionality of the system influences the stakeholder model. Had the system a function to explicitly support cleaning (e.g. a button to activate a cleaning mode of the machine), the `Cleaning Personnel` would become an actor of the sausage machine.
In addition to what we could derive from the accident report mentioned above, we added some more context to make the example more lively. The `Materials Delivery`

is a stakeholder of the machine as it delivers goods that shall be transformed into a product by the machine. That role should be informed when the machine is defect, or if the amount of material delivered is too high or too low with regards to the production rate of the machine. The `Sausage Processing` is a role of somebody doing something with the produced sausages. An example for a specialization of that role (not mentioned on diagram) would be a cook creating a dish with the sausage. The `Sausage Consumer` is interested in the cleanliness of the sausage production process, that only ingredients are used that do not harm her health, and that the sausage is tasty.

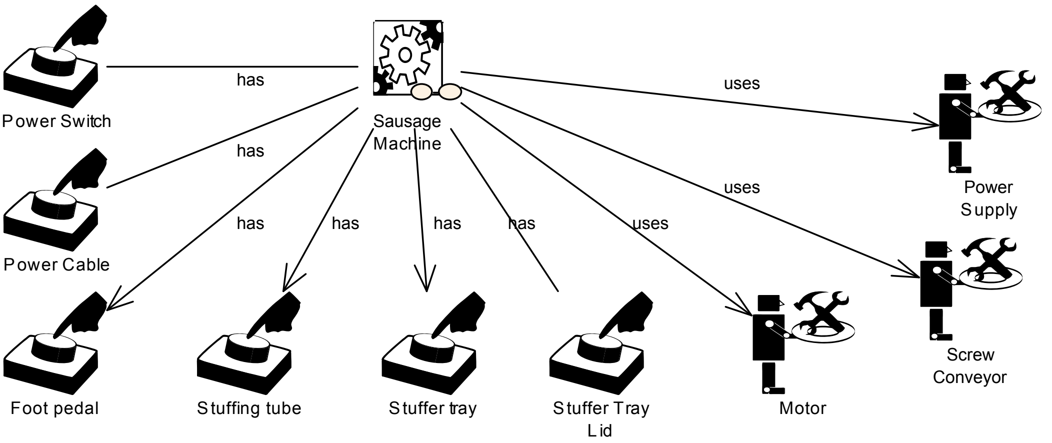


Figure 7.2.: Parts of the machine, internal and at the interface

From the accident report, we can derive the following parts of the machine as shown in Figure 7.2.[1] We start with describing the boundary objects. The machine is connected to the `Electricity Network` via a `Power Cable`. The `Machine Operator` uses more of the machine's boundary objects: He uses the `Stuffer Tray` to put material into the machine, the `Stuffing Tube` to wrap the sausage casing around it and handle the outcoming sausage. The `Footpedal` is used to start the operation of the machine. The machine uses some standard components that we model as service providers: The `Screw Conveyor` transports the material into the `Stuffing Tube` and supports regulating the amount of material flowing out of the machine. The Motor draws energy from the `Power Supply` and makes the `Screw Conveyor` turn. The `Power Switch` makes the machine operable, e.g. when it is turned off, pressing the `Footpedal` has no effect.

The sausage machine has one major function, `Create Sausage` . The `Machine Operator` puts a `Sausage Casing` around the `Stuffing Tube` and presses the `Footpedal` to fill the casing (Fig.7.3).

[1] To better understand which parts sausage machines usually have, we additionally consulted the internet sites of a sausage machine manufacturers and one distributor (Omcan 2014; Sirman 2014).

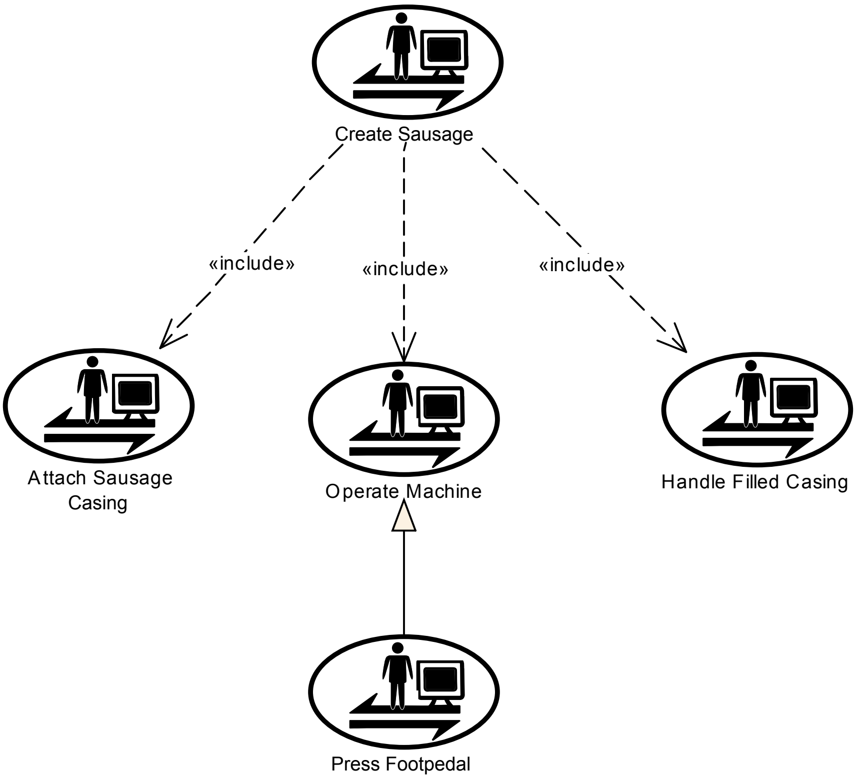


Figure 7.3.: Create Sausage sub-use cases

As a prerequisite, she should have filled the machine with `Sausage Material` via the `Stuffer Tray` (Figure 7.4), which is only possible with the tray lid removed. The lid should be fastened again afterwards.

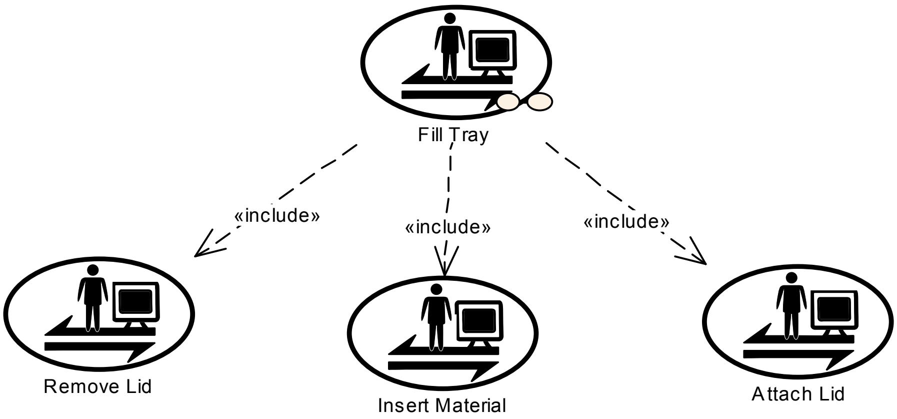


Figure 7.4.: Create Sausage sub-use cases

Before being able to operate the machine, she needs to use the `Power Switch` to `Turn On` the machine. Figure 7.5 shows the major use cases along with the participating actor and the boundary objects via with she interacts with the system.

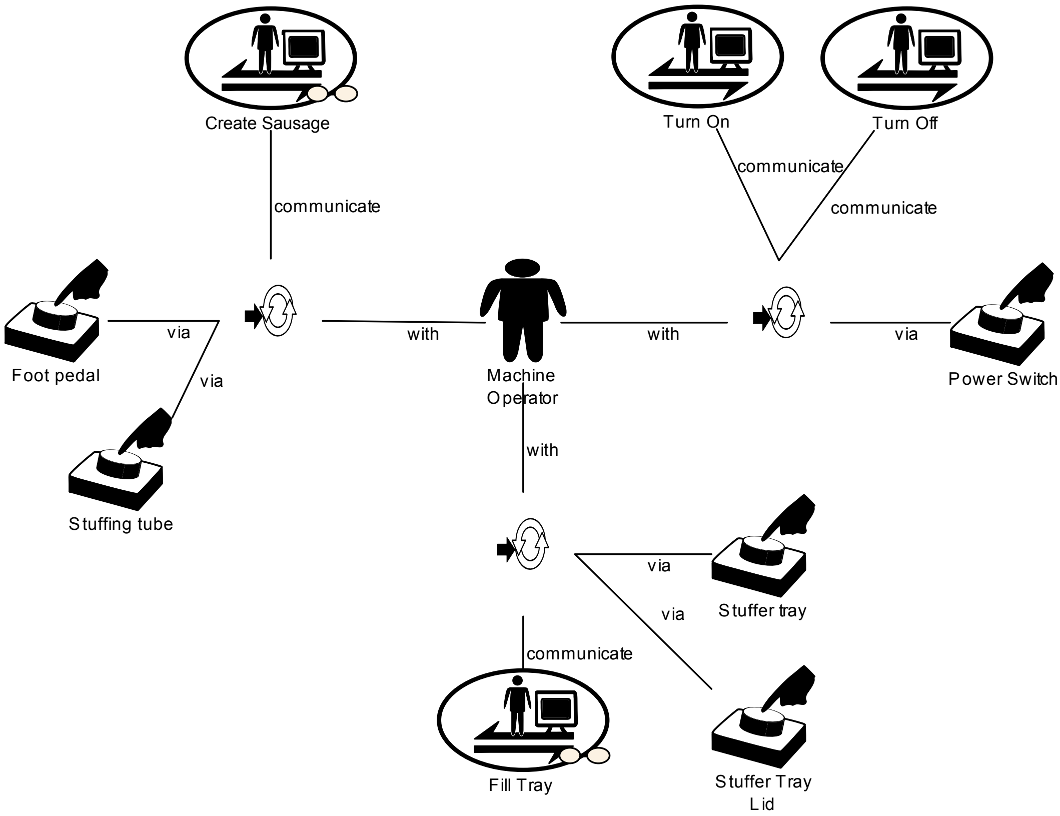


Figure 7.5.: Main use cases with actor and boundary objects

As Figure 7.5 is already crowded, we show the ordering of the use cases and the used and produced entity objects on a separate diagram in Figure 7.6.

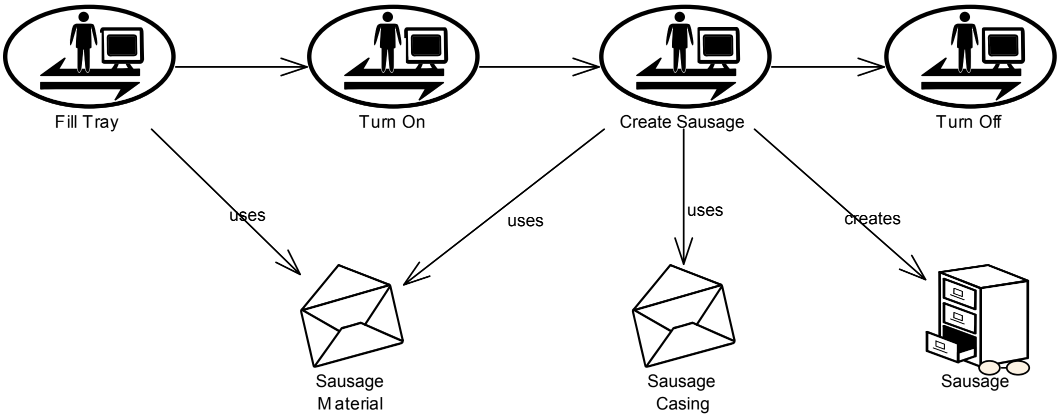


Figure 7.6.: Main use cases precedence and entity objects used and produced

On Figure 7.6 we already see that `Sausage` is a composite object. To clarify that it is composed of `Sausage Casing` and `Sausage Material`, we introduce Figure 7.7.

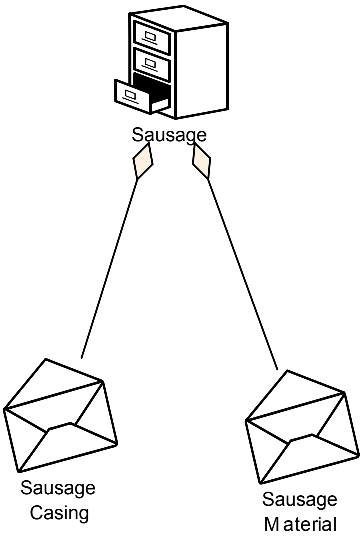


Figure 7.7.: Entity object decomposition

We continue with a simple hazard analysis of the system. From the accident report, we can learn that the `Screw Conveyor` should be guarded from unintended human interaction. For the `Machine Operator`, that means that she should not put material into the `Stuffer Tray` while the `Screw Conveyor` is turning, i.e. while the `Footpedal` is being pressed. With our process model from above, we deduct that the `Create Sausage` use case is vulnerable to `Injury of Operator` (See Fig. 7.8), which would harm the `Machine Operator` in case of incidence. It seems the machine manufacturer assumed the problem should be solved procedurally, as there are no parts of the machine avoiding or helping to avoid that somebody comes near moving parts of the machine while the machine is in operation. We can't tell for sure, as the accident report does not name the machine model and manufacturer. It is very likely that the operating manual of the machine has a security advice, as in the security instructions of (Talsa 2006, p15). We model such a security advice to keep anybody's hands away from the moving parts of the machine as a procedural mitigation. Furthermore, the employer of the `Machine Operator` should provide a security training, so that the operator can learn how to safely use the machine.

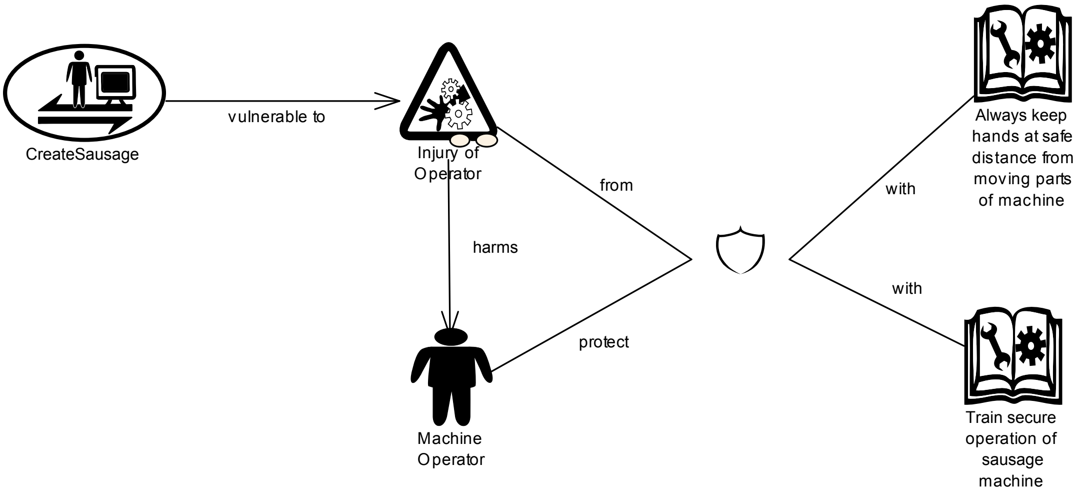

CreateSausage
vulnerable to
Injury of Operator
harms
Machine Operator
from
protect
with
with
Always keep hands at safe distance from moving parts of machine
Train secure operation of sausage machine

Figure 7.8.: Create Sausage vulnerability and mitigation of the hazard

As the Cleaning Personnel usually does receive such training, they should be advised to unplug the Power Cable before cleaning the machine (See Fig. 7.9). In the case of the accident, the worker in the role of `Cleaning Personnel` did not do so. According to the accident report (OSHA 2011), the employer was fined for violating paragraphs 3203(a), 5194(e)1, 3314(c), 4184(b), and 4185 of title 8 of the California Code of Regulations (OAL 2014)[2]. In detail this means that they had no "Injury and Illness Prevention Program" as required by §3203(a) and no "Written Hazard Communication Program" as required by §5194(e)(1). Also there was no sign to advise the worker that "[...] equipment capable of movement shall be stopped and the power source de-energized [...]" during "cleaning [...] operations" as required by §3314(c). 4184(b) together with 4185 would have required to install a guarding device on the `Footpedal` to avoid inadvertent operation. Sadly, all these procedural mitigations were in place and known, but the `Operating Company` failed to establish their realization.

[2] We are leaving out the regulations for which the employer was cited but not fined.

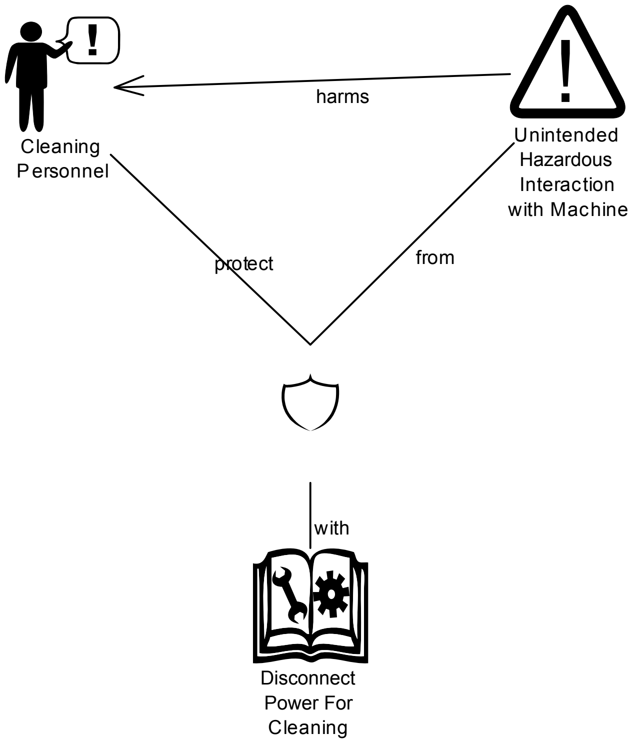

Cleaning Personnel
harms
Unintended Hazardous Interaction with Machine
protect
from
with
Disconnect Power For Cleaning

Figure 7.9.: Mitigation of hazard harming cleaning personnel

We summarize our danger model of the sausage stuffing machine in Figure 7.10. It extends the context diagram (Fig. 7.1) by adding the dangers that might harm the stakeholders. We already discussed how the operator and the cleaning personnel could be affected. In addition, the `Sausage Consumer` could suffer from a tainted sausage. The `Operating Company`'s fortune might be affected if the total cost of ownership (TCO; e.g. consisting of costs for personnel training, operation, maintenance, danger prevention, but also jurisdictional issues) of the machine gets too high. In the accident reported, the dangers affecting the Cleaning Personnel and the Operating Company became incidents: The cleaning worker's finger had to be amputated. The fines the employer had to pay certainly increased the TCO.

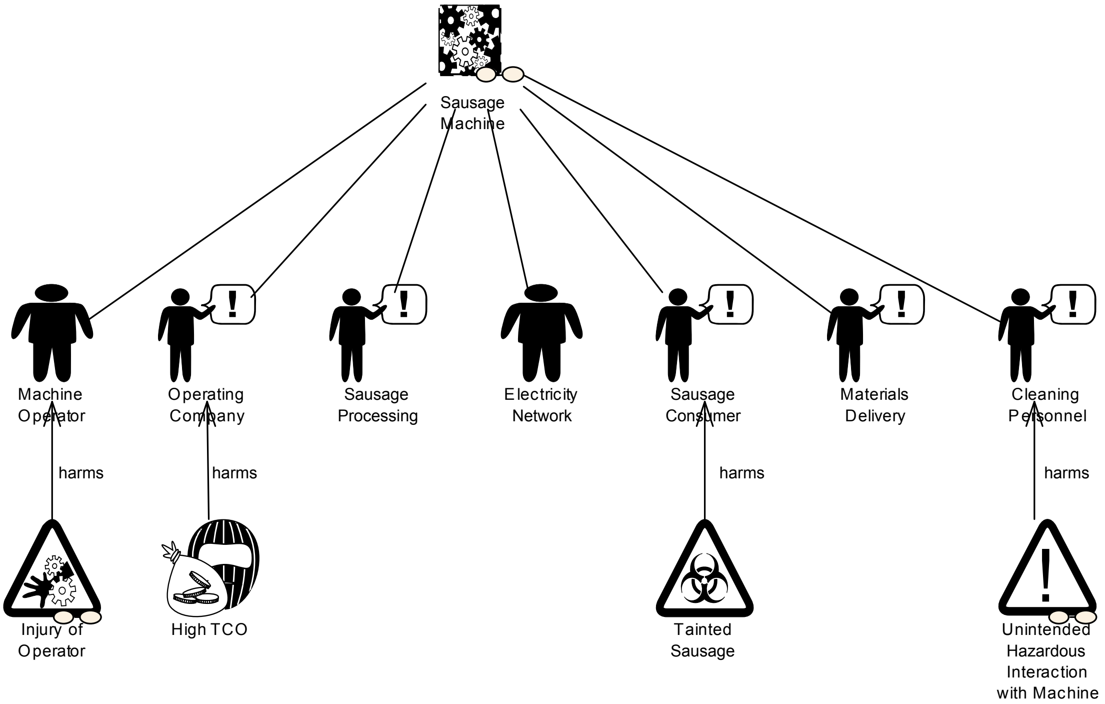


Figure 7.10.: Hazard summary with stakeholders

A functional requirement, that the `Footpedal` is ineffective as long as the `Stuffer Tray Lid` is not attached to the `Stuffer Tray`, could have saved the cleaning worker's finger. Even though there could be such a functional requirement mitigation of the dangers to `Machine Operator` and `Cleaning Personnel`, the machine manufacturer was legally not to blame, as the operating manual most certainly delivered the according safety instructions.

We deduct that procedural mitigations can be a weak spot of a system's design, because there is no way to effectively ensure that the knowledge present in the model is transported to the to-be-protected stakeholders. In order to achieve better protection, the `Operating Company` should have done some system requirements analysis: If we regard the system into which the sausage machine is embedded as the system under discussion, the sausage machine becomes a component in that system, supporting certain processes and being affected by others. In that system, we can trace features of the system to requirements that would be mitigating the dangers. Warning signs or a guarding of the `Footpedal` would be features of that system. From a modeling viewpoint, this would resolve the danger presented by the system. However, executives of companies operating potentially dangerous machinery would have to change their mindset, considering their company as a system-of-systems (SoS). Maybe then hazard analysis would lead to an analysis of possible resolutions by requirements which in turn would lead to a safer design of the SoS.

URML supports the first two parts of such a process. In addition, the above example suggests that URML can be used for domain engineering, and in particular can help regulators with creating graphical models of the safety requirements of a certain domain. In well-known domain engineering methods as FODA (Kang et al. 1990), the domain would be modeled in terms of features. URML supports additional requirements engineering concepts, all of which can support domain analysis. In the example above, we can see several generic elements that can be specialized differently for different systems. In the case above, the stakeholder `Sausage Processing` should be specialized by a stakeholder `Cook` but in a different setting it might be `Packaging Personnel`. The boundary object `Motor Activator` was specialized as `Footpedal` but also could have been a `Knee Lever`. The use case initiated by these boundary objects, `Operate Machine`, could respectively be specialized to `Press Footpedal` or `Press Knee Lever`. On such a domain model, regulators can check whether their regulations appropriately match the domain they regulate.

7.1.2. Clinical Chemistry Laboratory

A large part of this example model has already been presented in chapter 4 to showcase the notation of URML. In this section, more diagrams are added in order to provide context for these model excerpts. Some of the diagrams presented have been published in (Berenbach, Rea, and Schneider 2013). As stated in that publication, the example model has been developed with an expert of Siemens Healthcare Diagnostics. The creation process of the model has been process-driven, which means the expert first provided an end-to-end description of a typical phlebotomy process in a hospital. This description, along with some requested supplementary material was translated into a URML model which was iteratively reviewed together with the expert and enhanced where needed. It turned out the model has to describe a system-of-systems: the end-to-end process involves doctors, patients, staff of the hospital, the chemistry laboratory, and finally a blood analyzer machine.

Figure 7.11 provides an overview of the modeled process on multiple levels of abstraction. It also illustrates how diagrams are linked in the reference implementation of URML: A goggle symbol indicates that a diagram element is linked to another diagram. In the figure, the system under discussion is `Hospital`, shown with two use cases that it offers. The `Surgery` use case is not detailed. URML currently has no way of expressing out-of-scope elements, therefore the element has been marked as 'under construction' as a workaround. `Blood Sample Management Process` is detailed on the next level showing that it can be characterized with three phases. Each phase is then detailed on lower-level diagrams. In the figure, details of the `Pre-Analytical Phase` are shown: the diagram shows the steps starting from `Physician orders test for patient` to `Sample Preparation`. This workflow already crosses the boundary between systems: `Sample receipt in lab` is a use case of the chemistry laboratory. The diagram linked to `Sample Preparation` shows a different kind of diagram, expressing vulnerabilities of the use case, and a mitigation via a functional requirement and a procedure.

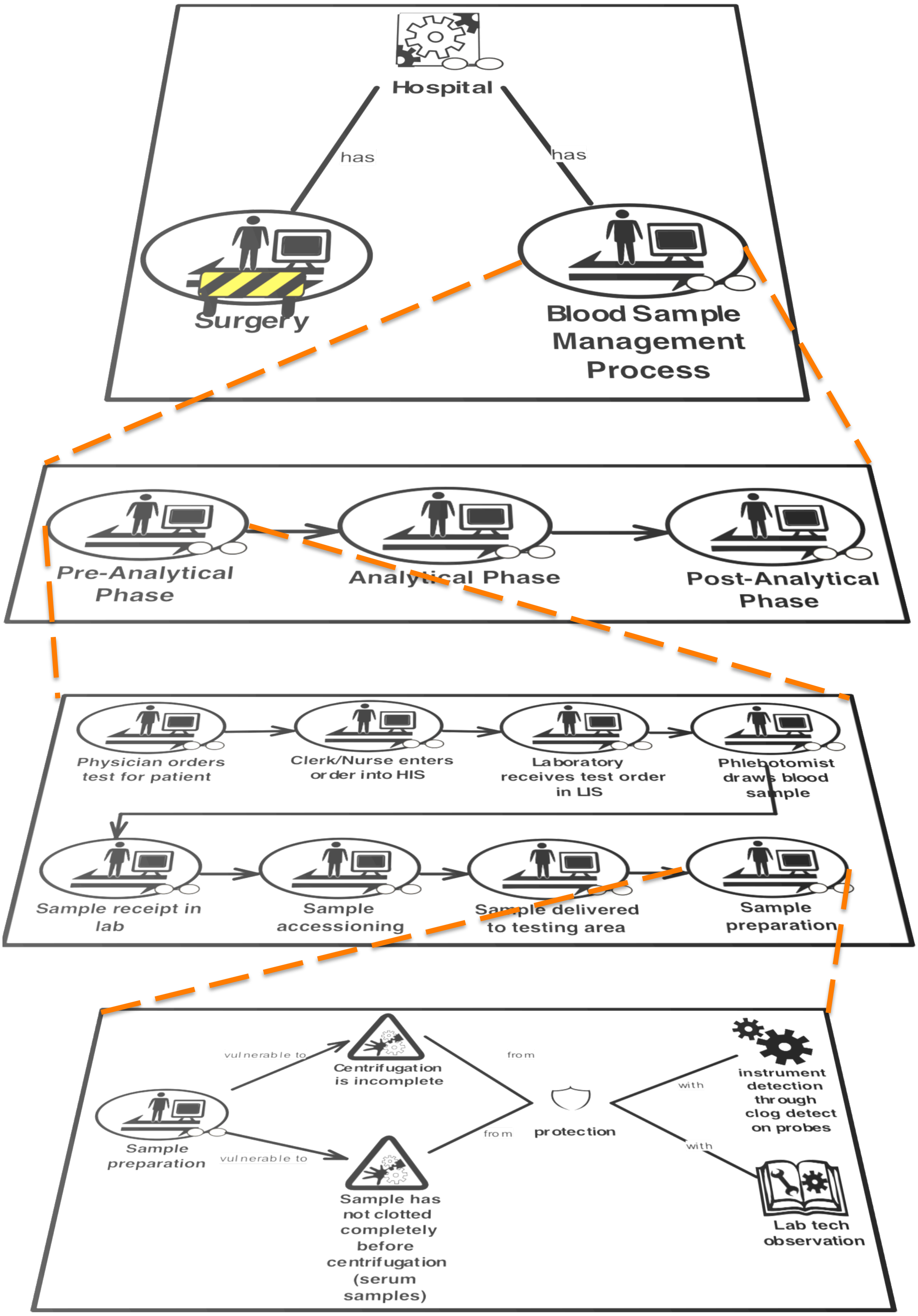

Hospital
has
has
Surgery
Blood Sample Management Process
Pre-Analytical Phase
Analytical Phase
Post-Analytical Phase
Physician orders test for patient
Clerk/Nurse enters order into HIS
Laboratory receives test order in LIS
Phlebotomist draws blood sample
Sample receipt in lab
Sample accessioning
Sample delivered to testing area
Sample preparation
Sample preparation
vulnerable to
vulnerable to
Centrifugation is incomplete
Sample has not clotted completely before centrifugation (serum samples)
from
from
protection
with
with
instrument detection through clog detect on probes
Lab tech observation

Figure 7.11.: Highlights on parts of the hierarchically composed `Blood Sample Management Process`

Due to the complexity of the model, the full model cannot be presented in this section. To achieve a broad coverage of the model, excerpts touching upon the analytical and post-analytical phase are presented next, as excerpts of the pre-analytical phase are already part of the figure above.

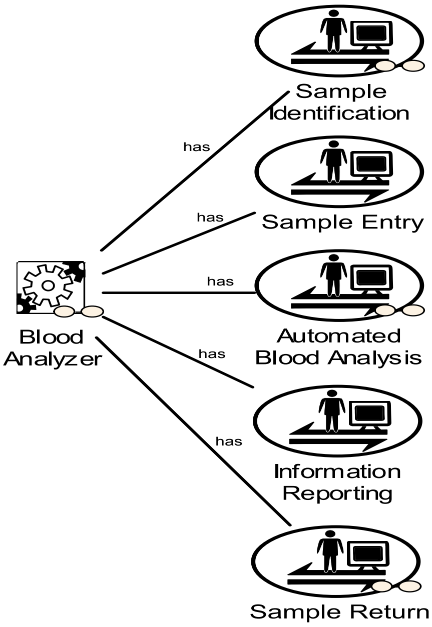


Figure 7.12.: Use cases of the analyzer

Figure 7.12 presents the use cases of a blood analyzer device. Users interact with such a system upon `Sample Identification`, `Sample Entry`, `Automated Blood Analysis`, `Information Reporting`, and `Sample Return`. The first step performed by the analyzer when doing the analysis is to check the integrity of the entered samples. Figure 7.13 gives an impression of the richness of abstractions included in URMLs meta-model. A functional requirement of the `Check Sample Integrity` use case is `Check samples for hemolysis`. It is part of a feature called `ASTM Hemolysis test`, which is one realization of the hard goal that the system shall achieve reliable hemolysis testing. Hemolysis is a medical term defined as "the breaking down of red blood cells with liberation of hemoglobin" ("hemolysis" 2016). A reliability quality requirement is therefore also part of the feature. The functional requirement is part of a requirements mitigation that protects the `Patient` stakeholder from distorted test results.

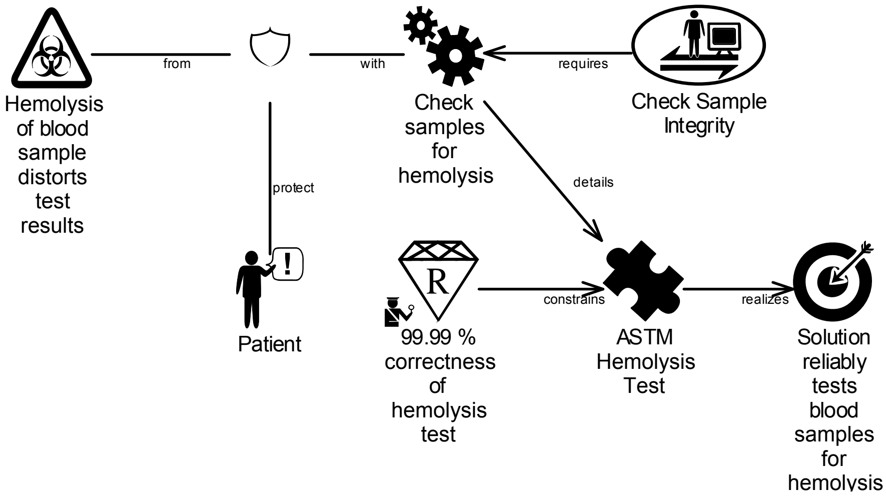

Hemolysis of blood sample distorts test results
from
with
requires
Check samples for hemolysis
Check Sample Integrity
protect
details
Patient
99.99 % correctness of hemolysis test
constrains
ASTM Hemolysis Test
realizes
Solution reliably tests blood samples for hemolysis

Figure 7.13.: Cross-sectional view on sample integrity aspects

Figure 7.14 deals with a use case of the chemistry laboratory, `Sample Archival`. The figure illustrates how an informal stakeholder request, `Easy Sample Handling`, was integrated into the model. After gaining further insight, the analyst has understood that there are two goals that have driven the stakeholder request: archived samples shall be easily available for retesting and additional tests. On the requirements side, the request is reflected by two requirements, one for the required functionality, and one for the required usability.

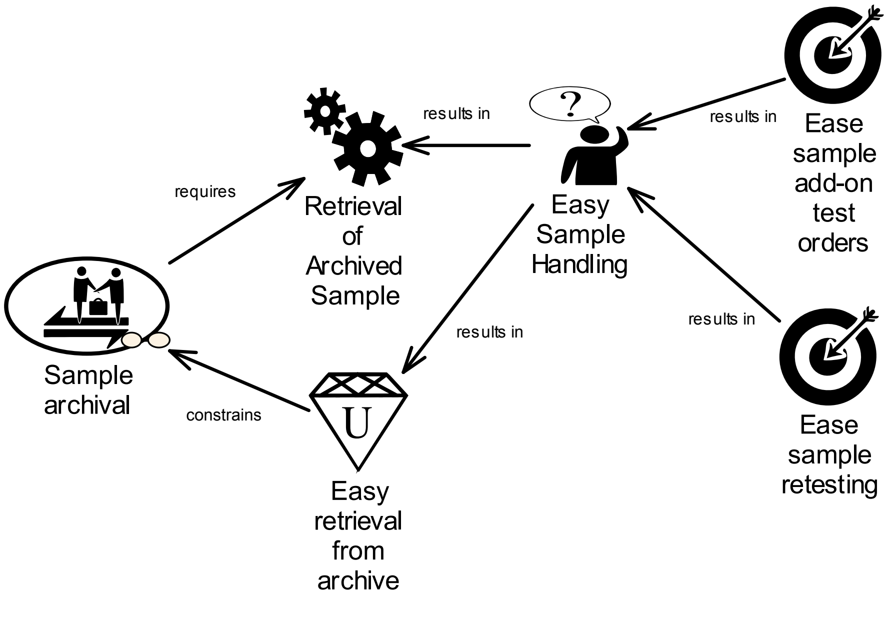


Figure 7.14.: Integration of an informal stakeholder request into the model

7.1.3. CMA@RE reference problem : bCMS

The “Barbados Car Crash Crisis Management System” (in short bCMS) software product line was published as a “focused case study” (Capozucca et al. 2012) for the CMA@RE workshop (Moreira, Ana, Georg, Geri, and Mussbacher, Gunter 2013). We created a URML model for the workshop. In this section, we provide a detailed description of that model. A shorter description is provided in the paper we submitted to the workshop (Schneider, Bruegge, and Berenbach 2013b). The case study simulates a scenario in which a “requirements document” is created by the customer and given to the development company. That requirements document provides a mostly textual description of the requirements towards the bCMS system and its variants (Capozucca et al. 2012). The customer knows some requirements terminology and uses the terms stakeholder, objective, functional requirement, and non-functional requirement. The case study is focused as only one top-level use case of the system is described in the document, along with some “static and dynamic variations”. The use case is about the coordination between a police and a fire station, both of which have been independently notified of a crisis. The emergency report of potentially multiple witnesses to each of the stations is outside of the scope of that use case. Also, it is assumed that the policies governing the measures and actions to be done to react to the crisis do exist and are known by the coordinators in both stations. We first present excerpts of the requirements model regarding the basic variant named `bCrash System`, and in later paragraphs show excerpts to illustrate variations. With the bCMS example problem modeled in KAOS (in Section3.1.6) and URN (in Section 3.2.3), the languages can be compared with URML on the basis of one consistent example.

The two roles of persons using the bCrash System are `Fire Station Coordinator` and `Police Station Coordinator`, referred to as FSC and PSC in the following paragraphs. In addition to these two, other kinds of persons have an interest in the proper working of the system, even though they are not interacting with it directly. These include firemen, police men, witnesses, crash victims, and government agencies.

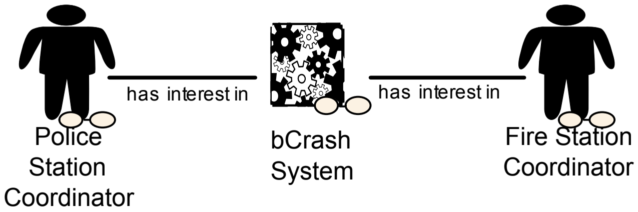


Figure 7.15.: Actors of the system

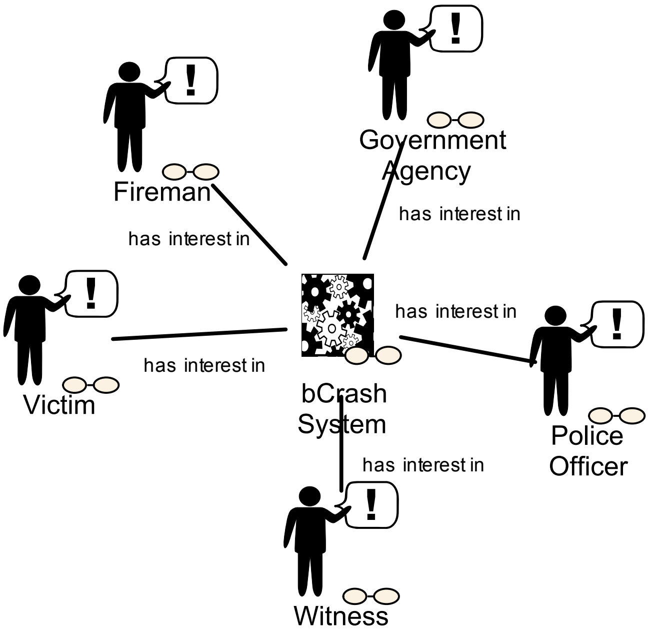


Figure 7.16.: Stakeholders of the system

We will focus on the main use case of the system, `Coordinate crisis management`. Both `FSC` and `PSC` need a dedicated interface that enables them to participate in the use case. The FSC can interact with the system via the `bCrash System Fire Station Interface` and the `PSC` with the according police-station-specific interface. Potentially the interface can be the same for both, but at this point in time we don't know that yet. Furthermore it is not unlikely that `FSC` and `PSC` need to specify different kinds of information via the interface.

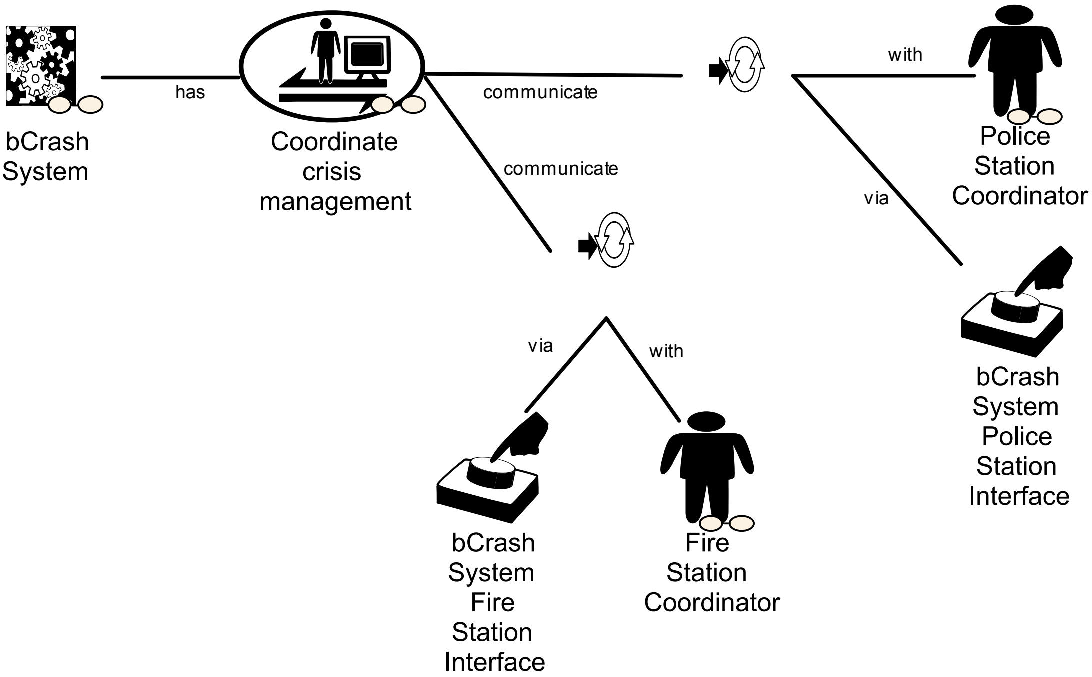

bCrash System
has
Coordinate crisis management
communicate
communicate
with
Police Station Coordinator
via
bCrash System Police Station Interface
via
with
bCrash System Fire Station Interface
Fire Station Coordinator

Figure 7.17.: Coordination of the management of the crisis is the main use case of bCMS.

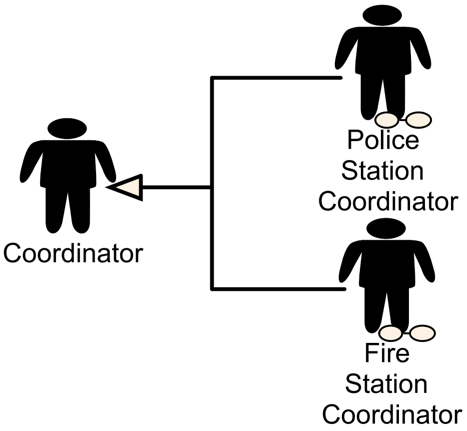


Figure 7.18.: Coordinator as superclass of PSC and FSC

Still, the two actors have much in common. From the textual requirements document, we can see that the two roles have identical objectives. So we can generalize the two into a more abstract `Coordinator` role (See Figure 7.18), and then show the goals that are common to both coordinators (in Figure 7.19). A coordinator wants to handle the crisis efficiently and effectively. We interpret this as the central goal of the coordinator, which is directly or indirectly supported by the other goals that he has. Getting resources to the crisis location in the shortest amount of time, having an accurate estimation of resource needs and time of arrivals for resources, having dependable communication with involved stakeholders, and providing clear and executable instructions to appropriate staff are all hard goals that have to be addressed by the system under discussion. We assume that their satisfaction can be measured by clear-cut criteria. The coordinator has two additional soft goals, for which we can't easily find clear-cut criteria: He wants to have effective negotiation skills (something the SUD can't address) and to maintain a feeling of control over the crisis (something the SUD can only partially address).

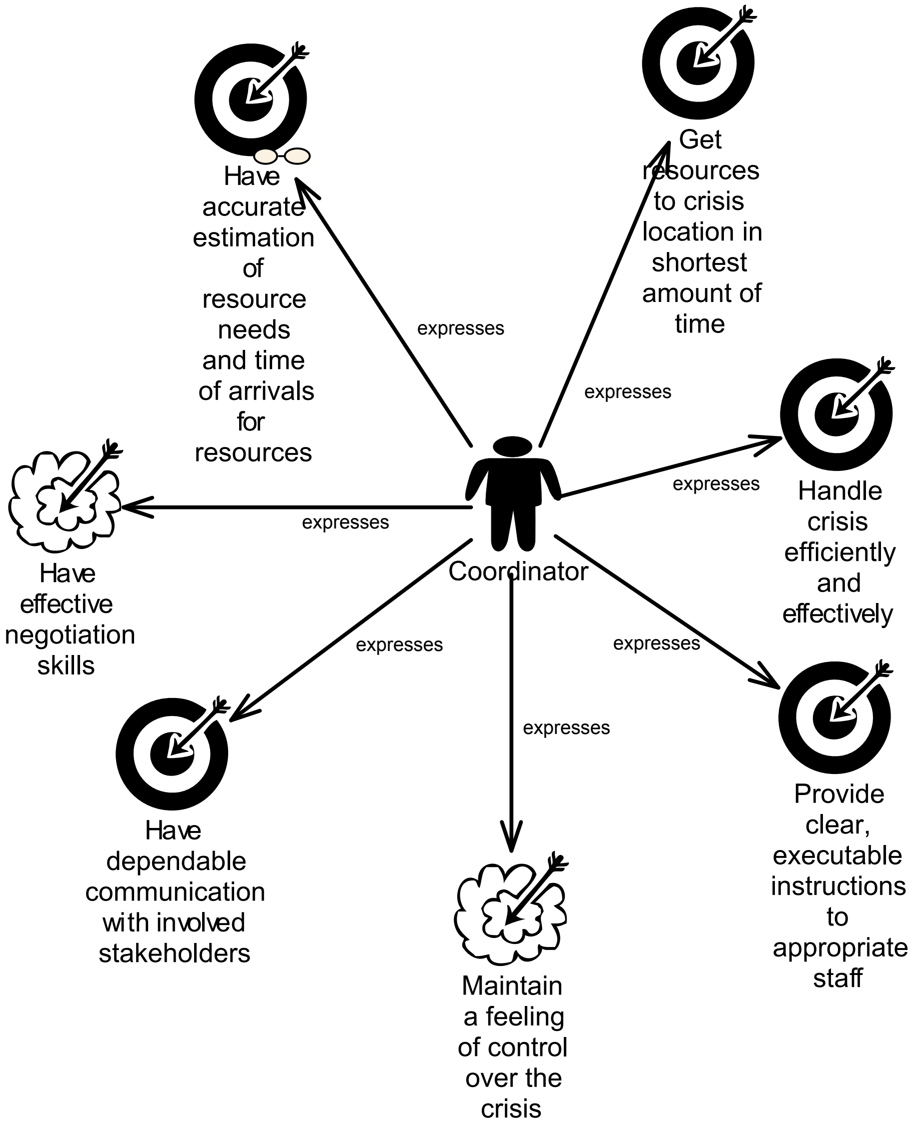

Have accurate estimation of resource needs and time of arrivals for resources
Get resources to crisis location in shortest amount of time
expresses
expresses
Handle crisis efficiently and effectively
expresses
Have effective negotiation skills
expresses
Coordinator
expresses
expresses
expresses
Have dependable communication with involved stakeholders
Provide clear, executable instructions to appropriate staff
Maintain a feeling of control over the crisis

Figure 7.19.: The goals that are shared by both kinds of coordinators

Inter-goal relationships are then visualized in Figure 7.20. The first soft goal, `Have effective negotiation skills`, supports two other hard goals, while the second, `Maintain feeling of control over the crisis`, supports one hard goal. That soft goal in turn has positive contributions from two other hard goals, which shows that the system will support the coordinator's feeling of control if it satisfies these hard goals.

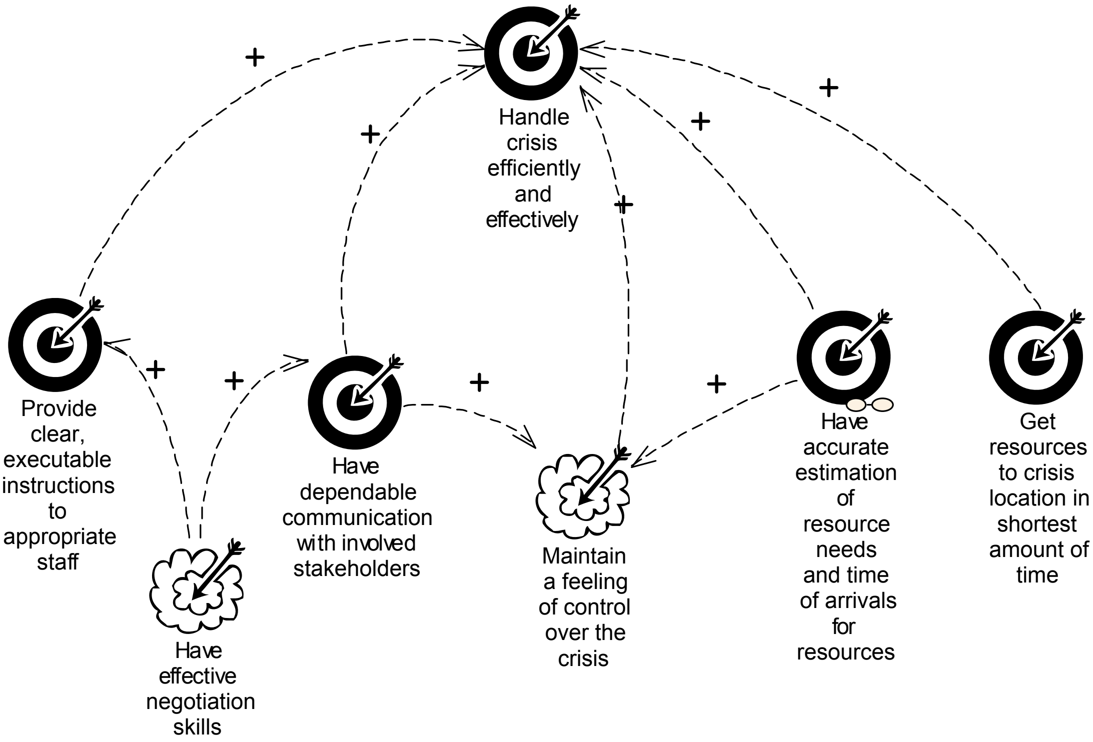

+
+
+
+
+
Handle crisis efficiently and effectively
Provide clear, executable instructions to appropriate staff
+
+
Have dependable communication with involved stakeholders
+
+
Have accurate estimation of resource needs and time of arrivals for resources
Get resources to crisis location in shortest amount of time
Have effective negotiation skills
Maintain a feeling of control over the crisis

Figure 7.20.: Contribution relationships between the coordinator goals

The customer already provided a description of the crisis management coordination workflow. A coordinator using the system will first establish the communication, then identify himself, and after both coordinators are connected and authenticated, provide the knowledge he has got about the crisis. Afterwards the two coordinators will jointly develop a route plan. Figure 7.21 shows that the first three use cases are modeled as atomic, which means they are not decomposed any further in the given model. `Develop route plan` does not have such an adornment, which states that there are some included and/or extending use cases in the model. Furthermore, it has a different adornment indicating that there is a linked diagram showing details about this use case. After the coordinators have agreed to the route plan, the routes for fire trucks and police vehicles are defined. Then the coordinators notify each other about vehicle dispatch, vehicle arrival at target location, and completed vehicle objectives. Eventually, the coordinators have to agree to close the crisis.

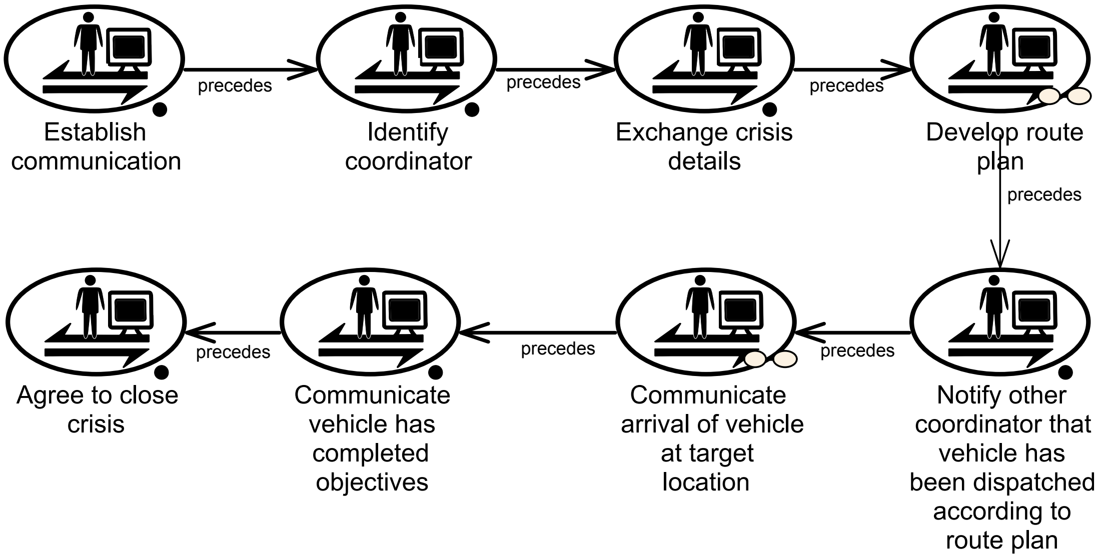

Establish communication
precedes
Identify coordinator
precedes
Exchange crisis details
precedes
Develop route plan
precedes
Notify other coordinator that vehicle has been dispatched according to route plan
precedes
Communicate arrival of vehicle at target location
precedes
Communicate vehicle has completed objectives
precedes
Agree to close crisis

Figure 7.21.: Coordinate crisis management workflow

Figure 7.22 illustrates how the `Develop route plan` use case is functionally decomposed. It also illustrates how URML basically reuses the UML use case diagram notation. Only the use case ovals are different, they aren't only oval shapes with text as in URML, but have icons inside.

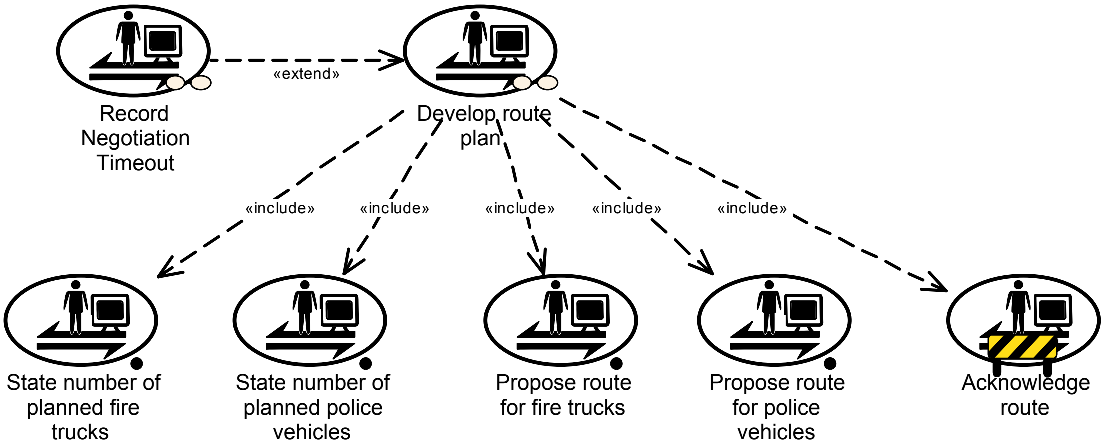

Record Negotiation Timeout
«extend»
Develop route plan
«include»
«include»
«include»
«include»
«include»
State number of planned fire trucks
State number of planned police vehicles
Propose route for fire trucks
Propose route for police vehicles
Acknowledge route

Figure 7.22.: Use Cases included in and extending `Develop route plan`

`Develop route plan` includes several other use cases. Both coordinators state the number of needed vehicles. Then the police station coordinator (not visible on diagram) defines the route for police vehicles, and proposes a possible fire truck route. If the fire station coordinator agrees, he may acknowledge the fire truck route. The `Record Negotiation Timeout` use case indicates that there is exceptional behavior: If any coordinator is taking longer than a pre-defined tim interval to respond to a system request, a timeout is recorded and the coordinators are alerted.

With URML, the constraint of a use case by a quality requirement can be shown. Figure 7.23 shows how the `Coordinate crisis management` use case is constrained by a reliability quality requirement. It states that in general, the data integrity of inter-coordinator communication shall be ensured 95% of the time coordination sessions are taking place. That quality requirement is refined into a more concrete requirement covering the data integrity of the crisis location data. The refining quality requirement is connected to an included use case which it applies to, `Exchange crisis details`.

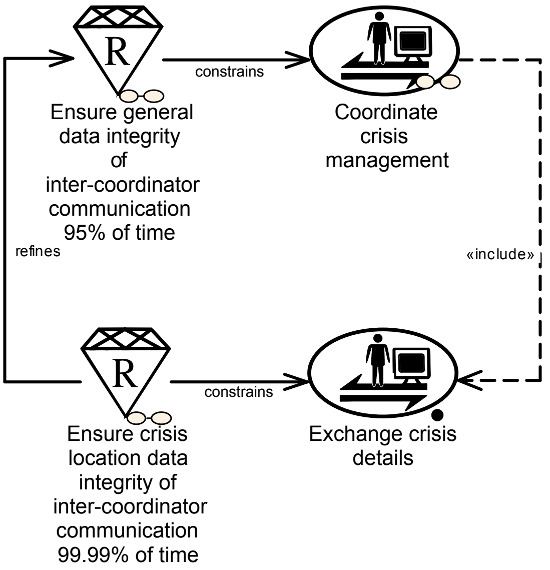

R
constrains
Ensure general data integrity of inter-coordinator communication 95% of time
Coordinate crisis management
refines
«include»
R
constrains
Ensure crisis location data integrity of inter-coordinator communication 99.99% of time
Exchange crisis details

Figure 7.23.: Quality requirements as use case constraints

The next two figures illustrate how URML supports the modeling of danger. In Figure 7.24, a hazard is related to the `Coordinate Crisis Management` use case via a vulnerability relationship. It indicates that the use case has a vulnerability that would harm a stakeholder on occurrence. If the coordination process is taking too long, that might have grave consequences for any victims of the accident. Therefore the bCMS system is required to alert coordinators to the fact they are taking too much time. This is modeled via a requirement mitigation relationship between the functional requirement, the stakeholder, and the hazard.

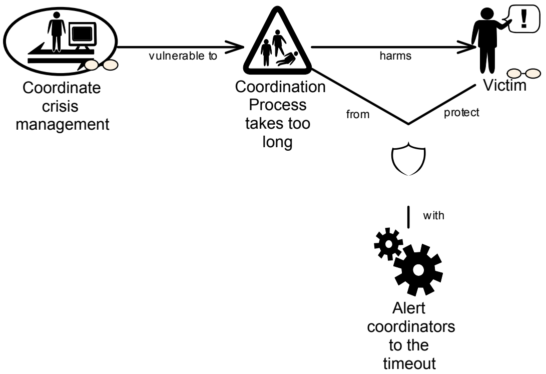

Coordinate crisis management
vulnerable to
Coordination Process takes too long
harms
Victim
!
from
protect
with
Alert coordinators to the timeout

Figure 7.24.: A hazard mitigated by a functional requirement

In Figure 7.25, a security concern regarding the `Identify coordinator` use case is visualized: Depending on the concrete implementation of the use case, there might be a threat of a man-in-the-middle attack. For example, if the identification is based on providing username and password, a third person having illegal access to these credentials might pretend to be one of the coordinators and thus is able to influence the coordination process without legitimation.

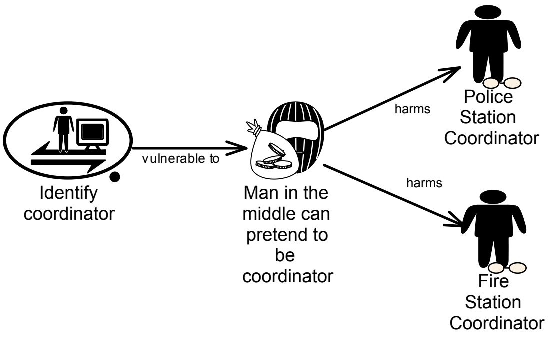

Identify coordinator
vulnerable to
Man in the middle can pretend to be coordinator
harms
harms
Police Station Coordinator
Fire Station Coordinator

Figure 7.25.: A vulnerability of `Identify coordinator`

The requirements document contains a lot of information about variants of the system for which the workflow has been described. All possible combinations of features are represented by a feature tree. Figure 7.26 shows the top level of that tree. It indicates that there is no variability regarding route plan management and progress reporting. Variable features include the communication channel, an optional vehicles management feature, support for multiple crisis, security, and the concrete communication protocol to be used by the system.

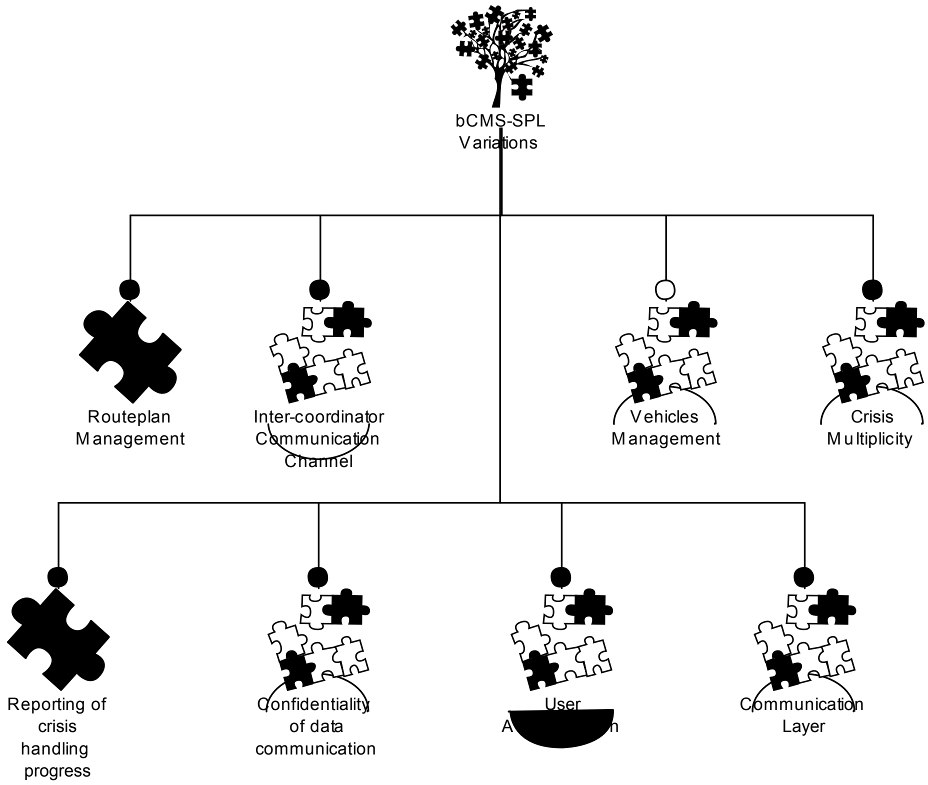

bCMS-SPL Variations
Routeplan Management
Inter-coordinator Communication Channel
Vehicles Management
Crisis Multiplicity
Reporting of crisis handling progress
Confidentiality of data communication
User
A
n
Communication Layer

Figure 7.26.: Top level of the feature tree describing all possible bCMS variations

The leaves of the feature tree are the potential features of the system under discussion, depending on the rules enforced by the feature groups they are part of. To indicate which features end up in a planned product, the feature list relationship is used to relate features with a product.

Figure 7.27 shows a simple product named `bCrash System` that has been derived from the product line. It supports only a one-to-one communication channel, the handling of a single crisis at a time, route plan management and progress reporting and will use the HTTPS protocol for the communication layer. No vehicles management is supported. This is conforming to the product line configuration rules shown in the figure above: `Vehicles Management` is an optional part of the product line. However, the modeler has forgotten about two security features of the system. Such errors could be avoided with tools implementing model checks that can be executed on demand or continuously.

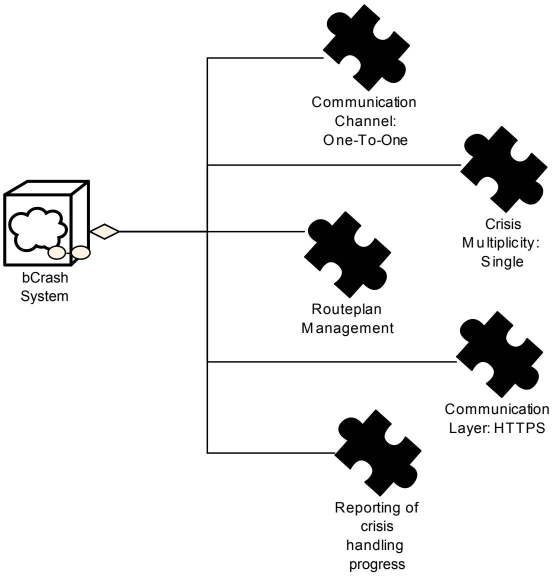

Communication Channel: One-To-One
Crisis Multiplicity: Single
bCrash System
Routeplan Management
Communication Layer: HTTPS
Reporting of crisis handling progress

Figure 7.27.: List of known features of a simple bCMS variant named bCrash

The complete feature tree of the bCMS product line is too large to fit on a page in well-legible form. It is presented anyway in Figure 7.28 to give an impression why modeling languages and implementing tools need complexity management mechanisms. The figure not only contains the feature tree, but also the product and its relationships to the leaf features from above. Even though the diagram is hard to read, it becomes apparent that the planned product (at the bottom) does not pick features from two mandatory feature groups.

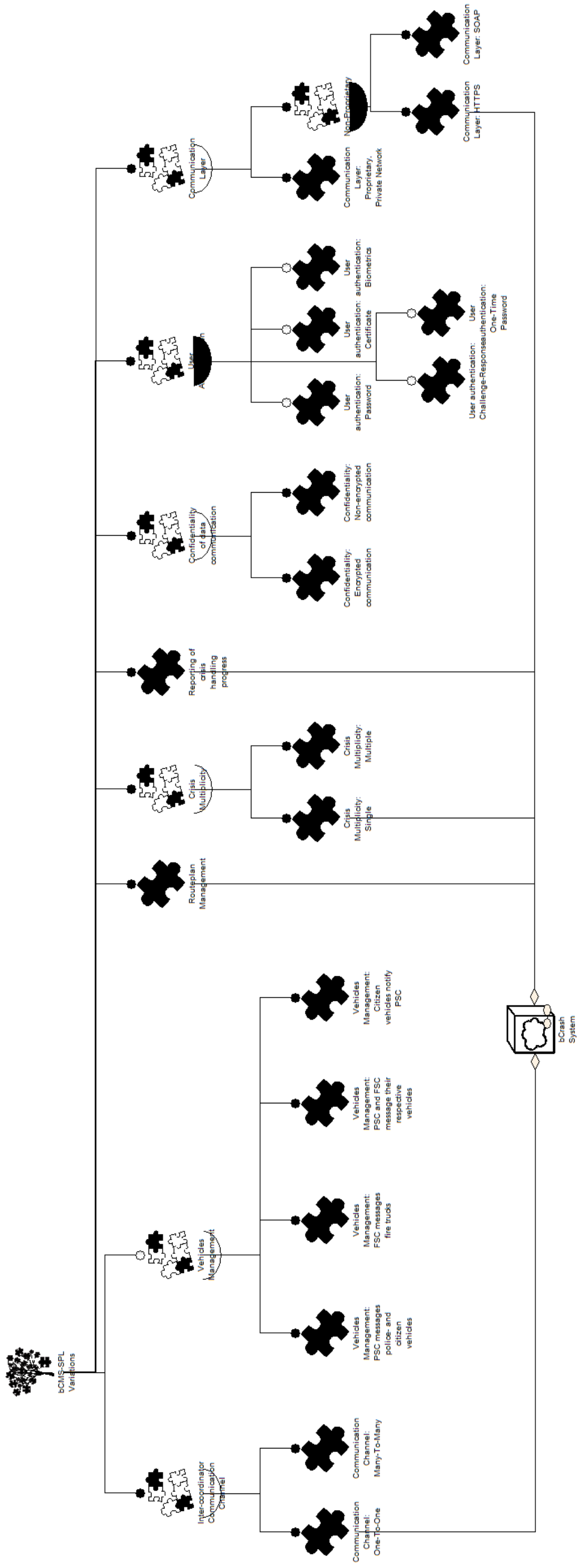

bCMS-SPL Variations
Inter-coordinator Communication Channel
Communication Channel: One-To-One
Communication Channel: Many-To-Many
Vehicles Management
Vehicles Management: PSC messages police- and citizen vehicles
Vehicles Management: FSC messages fire trucks
Vehicles Management: PSC and FSC message their respective vehicles
Vehicles Management: Citizen vehicles notify PSC
bCrash System
Routeplan Management
Crisis Multiplicity
Crisis Multiplicity: Single
Crisis Multiplicity: Multiple
Reporting of crisis handling progress
Confidentiality of data communication
Confidentiality: Encrypted communication
Confidentiality: Non-encrypted communication
User Authentication
User authentication: Password
User authentication: Certificate
User authentication: Biometrics
User authentication: Challenge-Response
User authentication: One-Time Password
Communication Layer
Communication Layer: Proprietary, Private Network
Non-Proprietary
Communication Layer: HTTPS
Communication Layer: SOAP

Figure 7.28.: Feature tree of a product line and a feature list of one concrete product on the same diagram

In Figure 7.29, the seven functional and two quality requirements that make the `Routeplan Management` feature are presented. The visual grouping of relationship lines by the modeler indicates that there might a need for another complexity handling mechanism. It seems the functional requirements are grouped into 'deal with the fire truck route' and 'deal with the police vehicle route'. With the current state of URML, the diagram could be improved by making `Routeplan Management` a feature group with two sub-features, one for each vehicle type. The corresponding requirements would then be connected to these new sub-features. This shows that URML inherently offers some complexity management, via the grouping of requirements in features.

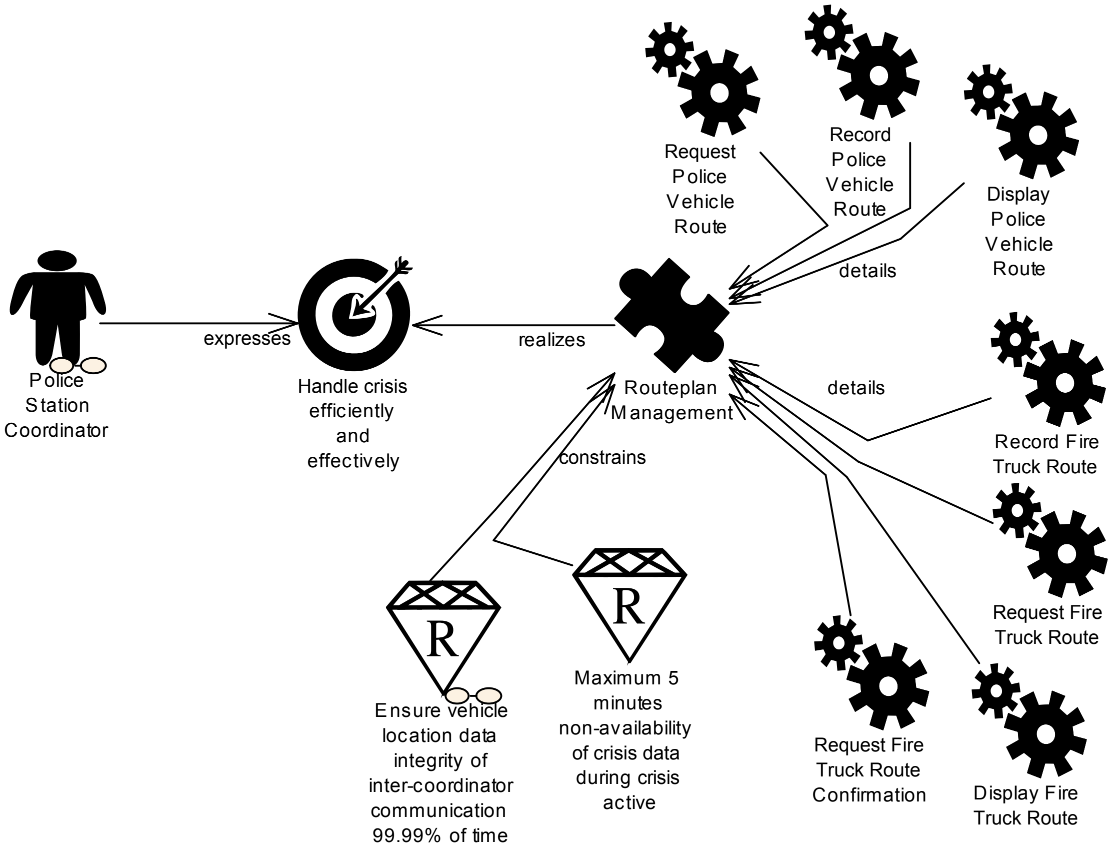

Request Police Vehicle Route
Record Police Vehicle Route
Display Police Vehicle Route
details
Police Station Coordinator
expresses
Handle crisis efficiently and effectively
realizes
Routeplan Management
details
Record Fire Truck Route
constrains
Request Fire Truck Route
R
R
Ensure vehicle location data integrity of inter-coordinator communication 99.99% of time
Maximum 5 minutes non-availability of crisis data during crisis active
Request Fire Truck Route Confirmation
Display Fire Truck Route

Figure 7.29.: Goal, feature, requirements

7.2. User Experiment: Suitability of Entity Icons

This chapter describes an experiment that evaluates the visual notations of URML, URN and KAOS. The experiment evaluates the symbols that are used to represent instances of entity meta-classes on diagrams. We measure how many errors are done in a task that entails mapping the various symbols of a visual notation to the corresponding term derived from the abstract syntax meta-model. We then observe how the error rates evolve over three repetitions of that task. A detailed description of the mapping task and the design of the experiment is given in section 7.2.1. The goal of the evaluation was twofold: First, we wanted to get an impression of the learnability of the languages. “Bad Learnability” would be concluded from high error rates that only decrease slowly over the task repetitions. Second, we strived for an experimental validation of which symbols are “badly designed”. Symbols with bad performance in the experiment (i.e. that are often matched to the wrong term) are hard to comprehend and to learn and thus are candidates for being redesigned or at least altered. These and other research questions are explained in section 7.2.2.

We assume that any of the compared languages can be learned and mastered by a requirements analyst given the proper training and continuous practical experience with the language. A language with bad performance in our experiment might be somewhat harder to learn than a language with good performance. We do not expect that a bad notation can make such a grave difference that it renders the language unlearnable, but we believe it might explain at least partially why the language has no broad dissemination in industry. There still is a controversy whether a visual requirements modeling language can be used for communication with customers. We believe that the design of the visual notation, and the appropriateness of the notation for the purpose of communicating with customers should be taken into account. We argue that if there was a visual requirements modeling language that was suitable for customer communication, its notation would perform well in our experiment. As should become apparent from the description in section 7.2.1, our experiment should measure how accessible the notation of the tested languages is to persons with only casual contact to the language. We claim that domain experts that would participate in requirements elicitation only under rare conditions are trained in requirements modeling languages. Therefore we assume that in the scenario we are discussing, domain experts would be persons with casual contact. Sessions with such persons would become tedious if the requirements analyst has to explain the symbols used on his diagrams again and again. We further argue that if the requirements analyst can easily support elicitation sessions with diagrams expressed in a language, it will be much more likely that the requirements analyst will also use the language for analysis. If the language was only used internally (i.e. only within the consulting or solution providing company) but the models are never presented to the customer, we believe that it will be much harder to validate the models with customer. A transformation to another format will become necessary and from this the question will evolve why requirements analysis is not altogether done with languages and tools supporting that format.

In the following sections, we will first give a detailed description of how we designed the experiment, then we report on its executions and present a summary of the collected data. This is followed by an interpretation of the collected data.

7.2.1. Experimental Design

The basic idea of the experiment is a mapping task: A symbol has to be mapped to the corresponding term (See Fig. 7.30).

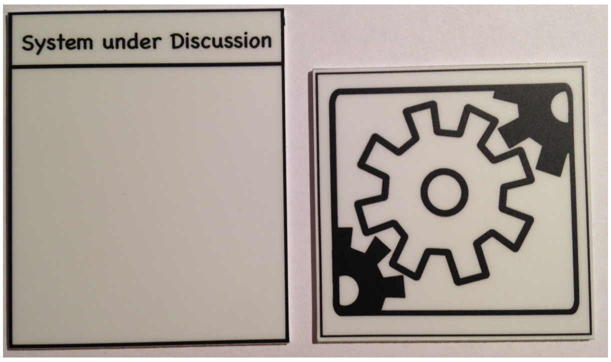


Figure 7.30.: Name (left card) to be matched to icon (right card) or vice versa

For the matching task, we have prepared plastic cards, divided into two groups. The cards of the first group hold the symbols with which URML entities would be represented on diagrams. The symbol is defined by the notation of the language. Most often, one meta-class corresponds to a certain symbol. In some cases, attribute values modify such a symbol or lead to an entirely different symbol. In such cases, we prepared a dedicated pair of cards (See Fig. 7.31 for an example).

On the cards of the second group, the terms corresponding to the symbols (the meta-class name and, where necessary, a term describing the variation, e.g. "Leaf" as a prefix to "Environment Process") are printed (See Figures 7.30 and 7.31). For a given meta-class, however, we did not print as many cards as possible combinations of attribute values would have suggested. As an example, we discuss the `UseCase` meta-class. `UseCase` has five attributes which it inherits from `Process`. `preCondition`, `postCondition` and `businessProcess` do not influence the notation, but `underConstruction` and `leaf` do. Both have boolean values, so there would be four combinations we could print a card for. We printed however only three cards, not taking into account the variant where both `leaf` and `underConstruction` are `true`. We argue that for the experiment, it is sufficient to test every overlay once for a given meta-class. Due to similar considerations, we did not print a card for every possible combination of `regulatory` and `type` of `QualityRequirement`. We have created 59 pairs of cards for URML, 18 pairs for KAOS, and 41 pairs for URN. For KAOS, there is no single authoritative specification, so we assembled the list of symbols from the KAOS tutorial (Respect-IT 2007) and van Lamsweerde's textbook (Lamsweerde 2009), the first using colored symbols, the second only grey toned symbols. For URN, we went through the chapters of specification that specified notation, page by page.

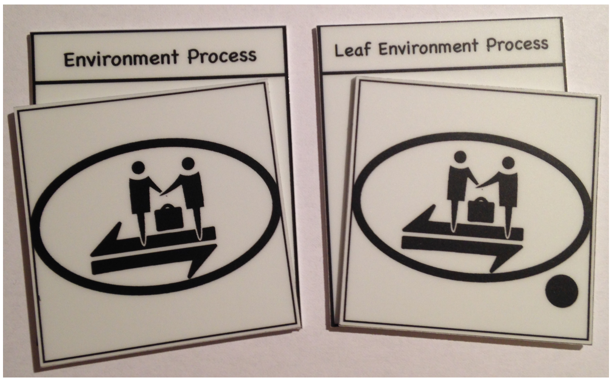


Figure 7.31.: Example with URML, where an attribute value would add an adornment to the base symbol

The experiment focuses on the entities meta-classes and their notation, not evaluating the relationship-centric part of the notation was a deliberate choice. Most relationships are depicted by a somehow structured line. The line is sometimes decorated with some visual or textual adornment, sometimes placed at the ends or sometimes in the middle of the line. Some notations seem to suggest a certain line routing style but do not formally define it. The degree to which textual labels are on relationship notations varies between the three languages. URML making heavy use of textual labels on relationship lines would have been favored by the style of the experiment: For example mapping a simple line labeled with "has interest in" is easily mapped to term "System Interest". For n-ary relationships, a decision is to be made with how many connections the relationship should be tested. It would have to be decided whether testing an n-ary relationship as a connection of three related elements would be more significant than testing it as a connection of four elements.

For a schematic of the overall course of the study, see Figure 7.32. The study consists of n experiments. Each experiment is to be conducted with one person, the participant, with a defined language under test. At the beginning the experiment, the participant is asked to answer a questionnaire. After the questionnaire, the experiment mainly consists of four executions of the basic idea. These four executions are however split into two parts. Part A consists of three consecutive runs. Part B consists of only one additional run.

Between Part A and Part B, there should be a time interval of at least three weeks, in which the participant should not be predominantly be occupied with learning the language. After each part, a short interview is conducted, allowing for feedback and observations of the participants. In the following paragraphs, we will first detail the questionnaire, then explain the different runs, and finally what is discussed in the interview.

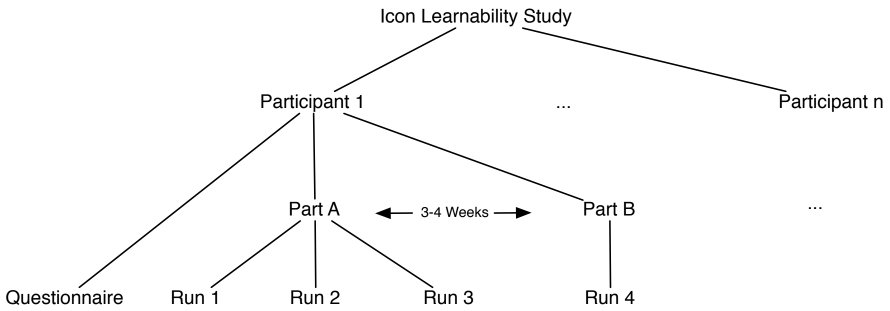


Figure 7.32.: Sketch of study design

Questionnaire In the questionnaire, participants shall first of all indicate their age and gender. We assume that age and gender do not have a direct impact on the results. Then we ask for a self-evaluation of the participant's english speaking skills on a five-valued scale from one to five. We assume that a value of five indicates the participant has English as one of her mother tongues. Participants that indicate a low value here are asked whether they want to use the german translation. The answer to this is also recorded. Participants with low fluency in English will at least need longer times for completing the experiment.

We then ask for a list of the participant's educational degrees. From this list, we will extract the following two bits of information: 1) Whether the participant has any IT background and 2) whether the participant has any academic background. We assume that participants with an IT background will make less errors as they will already know about some terms or icons presented in the experiment. We assume however that most participants will not be familiar with the tested notations. We will therefore also record whether the participant has some knowledge about the tested notation.

The following two questions aim at partially understanding the participants cultural background. We ask for the country in which the participant was born and which one she is currently living in. These data will not allow for any in-depth study of the influence of cultural background on the experiment results. We do record these values in order to explore whether follow-up studies should go into more detail here. If country of birth and / or country of residence would have a significant impact on the results, we would have a strong motivation for an in-depth investigation.

Runs For each of the four runs, the matching task is executed once. A single run is started with all icon cards lying shuffled and uncovered on a table. The term cards are arranged in a grid, without sorting. The term cards are shuffled by the experimenter prior to putting them on the table. Upon start of the run, the experimenter tells the participant to begin and starts taking the time. The run ends as soon as the participant has completed all pairs of cards and indicates she is finished. There is no time limit. After recording the time used for the run, the experimenter counts the numbers of correct and incorrect pairs, and records these numbers in a spreadsheet. Wrong numbers regarding incorrect matchings would distort the results. In order to avoid errors on counting the incorrect pairs, we separately count the correct pairs. A formula in the spreadsheet indicates whether the sum of correct and incorrect pairs equals the total number of pairs. Additionally, a photograph of the matched cards is taken. It serves as another control mechanism to avoid counting errors, and for post-experiment analysis which incorrect mappings happen more often than others. After the photograph, an explanation is given to the participant how the correct mapping could be memorized and a sample solution. The participant then still has some time to look at the sample solution. Then, the next run is started with shuffling of cards and laying out of the grid. For participants who have German as their mother tongue, we have prepared a translation of the english terms into German in tabular form, which a participant could use during the matching task. For any participant, we allowed a re-arrangement of the term cards, with the single requirement that the participant does not create the same custom layout on each run. That requirement should avoid that participants memorize correct mappings by location on the table.

Interview In the interviews concluding parts A and B, we first discuss two questions with each participant. The first question is “Which icons do you find difficult to match?”. The experimenter records the icons mentioned here. If participants have any remarks on the mentioned icons, e.g. some rationale why they find the icon irritating or misleading, this is also recorded. The second questions is “Which names do you find confusing?”. We expect that there will be a certain overlap between the answers to these questions. Complicated terms, are also often hard to visualize, and thus may likely have an unintuitive symbol. We also expect very generic terms that can have many different meanings, as e.g. “Object” will be mentioned here. We expect that most participants will have additional remarks, and thus ask as a third and last question “Do you have any other question or comment?”. The data recorded here might suggest changes that can be incorporated into future variants of the study. We expect that the data gathered in the interview of part A will partially be confirmed in the interview of part B. Should a participant explicitly indicate that an icon or term was not confusing her anymore, we would take it from the record.

Post-Experiment-Phase After the experiment, the experimenter uses each photograph to record the wrong pairs of each run. We will use this data to investigate which pairs were wrong most often. We assume that there are several reasons for wrong pairs:

- Meaning Proximity: Two concepts have a similar meaning
- Icon Proximity: Two concepts are depicted by similar icons
- Misinterpretation of Icon: The participant saw something different in an icon that seemed to fit to another term
- Misinterpretation of Term: The participant had a different understanding of the term and seemed to find a fitting icon

- Chance: The participant had no idea where to put the icon and just put it somewhere

In the case that we will observe a significant number of the same wrong pair over all runs of all participants, we will do an in-depth analysis, as far as our recorded data allows.

In addition, we will count how often a concept (i.e. its term or its icon) was not correctly mapped. Due to the nature of the experiment, if a term gets the wrong icon, its corresponding icon will have been put on another wrong term. Therefore, we do not count the error twice. In contrast to the previous metric, which could indicate which concepts are often confused, this will allow us to find out which concepts of a language are the hardest to map.

Furthermore, the photograph establishes quality assurance of the counts of correct and incorrect pairs: the experimenter should count the same number of wrong pairs as he did during the experiment.

The notes from the interviews should be summarized after all experiments are done. We can see then if multiple participants had the same observations or comments.

Primary Data collected For any participant we have recorded the answers to the questionnaire, i.e. age, gender, english skills, education, country of birth and country of residence. Furthermore we have the experimenter's notes from the interview. For any run i of a participant, we count the number of correct c_i and incorrect pairs e_i, and record the time used t_i. The correct pairs are counted only in order to have another control mechanism to avoid incorrect counts. The correctness of the counts can be determined by checking whether the sum of the counts for correct and incorrect pairs equals the total number of available pairs.

Derived Data The number of incorrect pairs of participant n at run i, $e_{i,n}$, divided by the total number of pairs p is the error rate $R_{i,n}$ of a participant n's run i. p depends on the language under test, $p = 59$ for URML, $p = 41$ for URN, and $p = 18$ for KAOS. We can state that the error rate is strictly decreasing between runs i and k if $R_{i,n} \leq \ldots \leq R_{k,n}$.

$$R_{i,n} = \frac{e_{i,n}}{p} \tag{7.1}$$

For the individual participant n we can compute the average error rate A_n.

$$A_n = \frac{R_{1,n} + R_{2,n} + R_{3,n} + R_{4n}}{4} \tag{7.2}$$

From the error rates of all runs i, we can compute the average error rate on run i, A_i, with N being the number of participants that tested the given language.

$$A_i = \frac{\sum_{n=1}^{N} R_{i,n}}{N},\ i\,\epsilon\,[1,2,3,4] \tag{7.3}$$

With the four A_i, we can derive the average error rate A over all runs of all participants.

$$A = \frac{A_1 + A_2 + A_3 + A_4}{4} \tag{7.4}$$

Relationship to related work Caire et al. have evaluated the research question whether naive users can design a visual notation with a higher semantic transparency than experts have done in the past (Caire et al. 2013). To evaluate the research question, Caire et al have conducted a four experiments and one nonreactive study, among which are a semantic transparency and a recognition experiment. The experiment presented above can be considered similar to the combination of these two experiments. The first run of our experiment is similar to the semantic transparency experiment, focusing on the immediate understandability of the symbols. Runs three to four are similar to the recognition experiment, where the focus is in how good the meanings of the symbols could be remembered.

The nine concepts that were evaluated in Caire et al's experiments (`Actor`, `Agent`, `Belief`, `Goal`, `Position`, `Resource`, `Role`, `Softgoal`, and `Task`) were chosen from i*, which had a strong impact on URN, the notation that we have evaluated. In addition some of these nine concepts were also integrated in URML or are present in KAOS. We will therefore relate to their results in subsection 7.2.3, wherever our results can be put in context.

When designing the experiment presented above, we had no knowledge about the experiments done by Caire et al. A striking similarity is that we have both attempted to evaluate the symbol set with “naive” users, i.e. users that had no previous experience with the language under test. Caire et al. added the additional constraint that the participants may not have an IT background, which we explicitly allowed. An important difference between the experiments is also that notation design was not under study in our experiment. We are predominantly interested in how to evaluate a given notation, not changing its design process.

We have also learned that our experiment design is very close to a guessability study (Wobbrock et al. 2005), where guessability is defined as “that quality of symbols which allows a user to access intended referents via those symbols despite a lack of knowledge of those symbols”. The guessability study described by Wobbrock et al. also shares some similarity with the one of Caire et al. as the symbols to be evaluated were also proposed by the participants of the study. The skepticism whether an optimal symbol set can be found in a small range of experiments applies here as well. In spite of this, we are convinced of the importance of guessability, or semantic immediacy, as Moody would phrase it, for the symbol design of a visual notation. We think that modeling language design should take more existing scientific knowledge from the field of human-computer interaction (HCI) into account.

7.2.2. Hypotheses

We are interested in how exactly the average error rates R_i change over the four runs. In general, we expect a decrease in the error rates from the first to the third run and a slight increase from the third to the fourth run. We also hypothesize that every language under test will have its characteristic “learning curve”.

We are also interested in observing whether the error rate consistently decreases for all participants, or if there are groups of participants that do perform differently than others.

A potential grouping of participants can be done with the data from the questionnaire.

An additional indication of a good learnability will be if, parallel to the error rate, also the average used time decreases. We do not want to over-emphasize this time metric though. With the given experimental design, we control not enough variables to thoroughly interpret the time metric. We assume most of the time for completing a single run will be used for searching a specific card, not for doing the mental mapping between name and icon. Also, differences in the completion times might not only be dependent on data covered in the questionnaire but on uncontrolled variables as time of day, lighting situation in the experiment environment, current concentration of the participant. Therefore, we do not average the completion time over all participants but instead investigate whether a decreasing completion time can be found for the majority of the participants. Nevertheless, if error rate and used time decrease consistently, we have a strong indication that the language is learnable. If the notation was too complex (e.g. because the icons are too hard to remember, or too easy to confuse), and thus hard to learn, we assume that the error rate would not significantly decrease but rather stay constant, fluctuate, or even increase. In detail, we want to investigate the following questions:

1. For each participant on a given language, how does R_i evolve?
2. For each participant on a given language, how does t_i evolve?
3. Will there be participants with $A_n >= 0.5$? i.e. will there be participants that on average produce more incorrect than correct results?
4. For all participants, does the average error rate decrease significantly between the first and the third run? How much will it increase on the fourth run?
5. For a majority of participants, does used time decrease significantly between the first and the third run? How much will it increase on the fourth run?
6. For a given language what is the average error rate over all runs? Will it be less than 0.5 for all three languages?
7. For a given icon of a given language, in how many percent of the runs was it part of an error?
8. Which icon of a given language has the most errors?
9. Is there a difference between IT experts and participants unrelated to IT ?

These are our expectations regarding the experiment:

- The average number of errors of a single participant will decrease over the experiment runs
- The average number of errors of all participants will decrease over the experiment runs

- the error rate of sibling icons will not be significantly higher than of other icons
- there is no icon in which a certain cultural background has a significant influence
- the performance is also independent of age or gender
- the performance varies depending on whether the participant has an IT Background or not
- URML will on average have lower error rates than the other languages, independent of background

7.2.3. Experiment Results

The study was conducted with 21 participants. Seventeen out of the twenty-one participants performed all four runs of the experiment, four participants only participated in the first three runs. The distribution of the participants to the three compared languages and how many completed the fourth run can be seen in Figure 7.33.

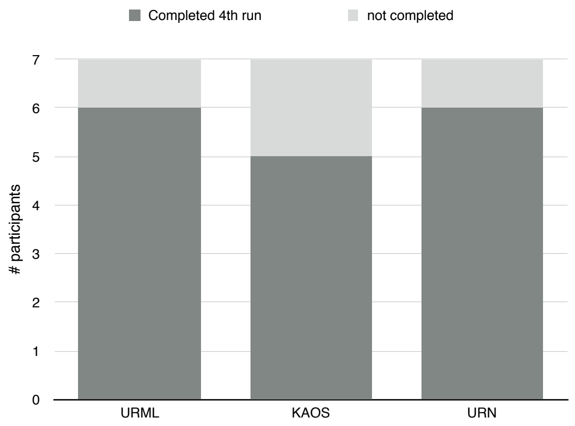


Figure 7.33.: Number of participants and indication how many completed the fourth run

By interpreting the total correct and incorrect mappings as a percentage of the total number of mappings, we can interpret the ratio of incorrect to total mappings as the average error rate of each language (Fig. 7.34). The total number of mappings results from multiplying the number of pairs with the number of runs and the number of participants (e.g. 59 x 4 x 7 for URML). URML has the lowest ratio of incorrect to correct mappings, in spite of it having the most symbols to test. KAOS in comparison has the highest ratio, even though it has the least symbols to test. URN's ratio is slightly below the one of KAOS but considerably above the one of URML.

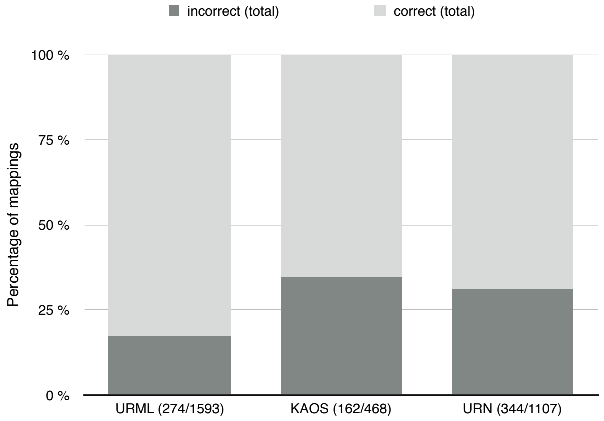


Figure 7.34.: The average correct / incorrect rates of the three languages. Category labels show the total error count versus the maximum possible errors.

Looking at the average error rates of the individual runs, we can see that for each language, the first run has the highest average error rate (Fig. 7.35). The rate then drops to near zero for the third run and goes up again for the fourth run. Despite that same general evolution of the average error rates, the different languages are very different in detail. URML has the lowest average error rate compared to URN and KAOS in the first run and the fourth run. For KAOS and URN, the delta between third and fourth run is much higher than it is for URML. KAOS has the highest average error rate in the first and fourth, URN has the highest in the second run.

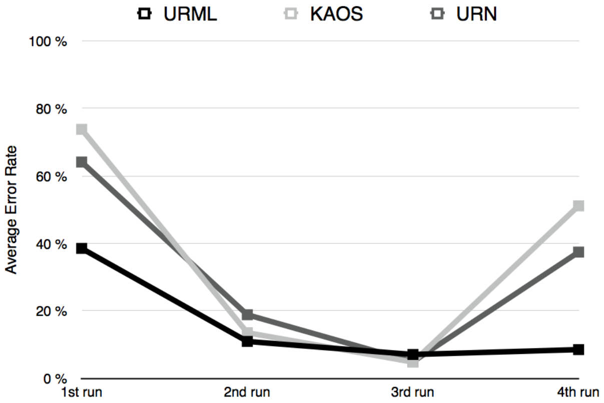


Figure 7.35.: The average error rates of the three languages on each run.

As there was no time limit for the participants, the question arises whether the participants on URML might have taken more time to complete the task and thus were more successful. As the different languages have different numbers of symbols, we can not compare the time used for each run. We therefore calculated a “seconds per mapping” value by dividing the time taken in seconds by the number of pairs. The average times per mapping can be seen in Figure 7.36. We can see indeed that on average, the participants on URML used more time to complete their task. This time difference however cannot explain the better performance of the URML participants, as all the other participants had the same freedom to take their time. All participants stopped the run as soon as they had no idea how to do any better. From our observations during the tasks, we can state that the notation of URML might have invited participants to put more thought into the mappings. This might be driven by the fact that in URML, there are families of symbols for which it is easy to draw analogies, for example for subclasses of QualityRequirement. This led participants to think that there should be more such systematic familiarities, which was indeed designed into the notation, for `Asset`, `Hazard`, `Process`, `Requirement`, and `ServiceProvider`. There were however symbols that had similarities in their symbols but shared no common meaning, which lead to some confusion.

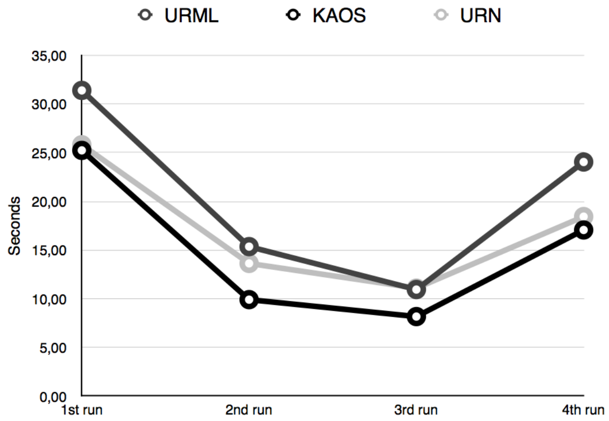


Figure 7.36.: The average times per mapping on each run.

Regarding the error rates of the fourth run, we can also look into whether the time intervals between the third and forth runs were considerably lower for URML, compared to the other two languages. The time intervals of the individual participants are shown in Figure 7.37. The average interval for URML is 37.5 days, for KAOS it is 47 days, and for URN it is 34.667 days.

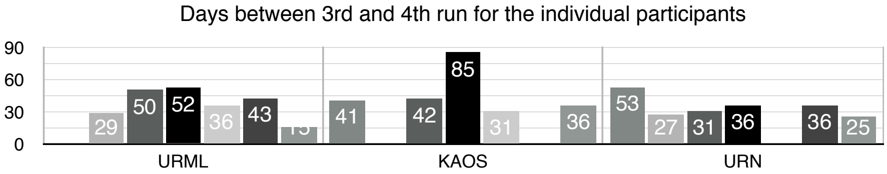
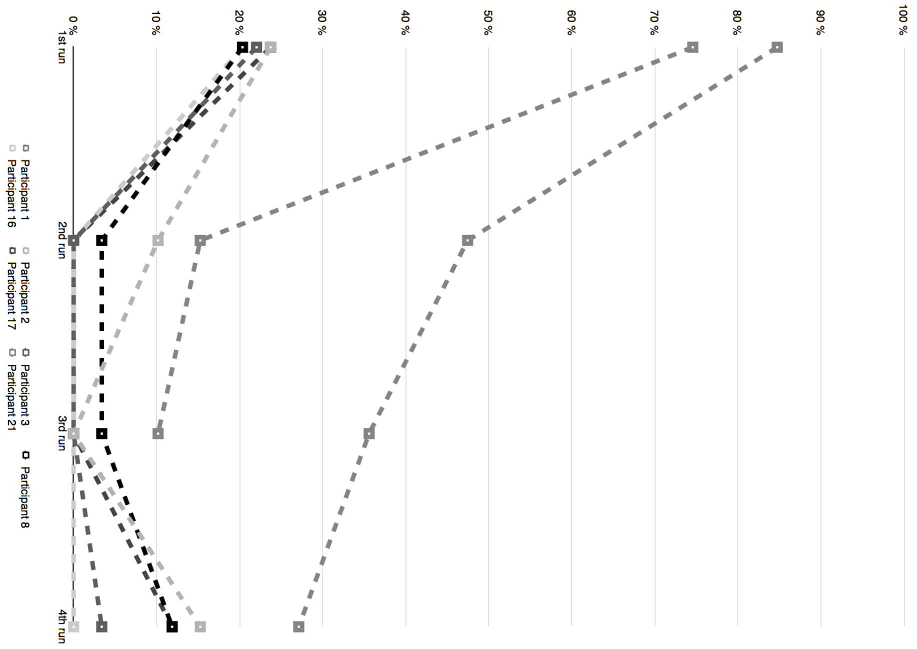


Figure 7.37.: Interval in days between the 3rd and 4th run of the experiments, for each participant. Gaps indicate that the participant did not do the 4th run.

For each language, we will now look into how the error rates of the individual participants evolved from run to run. For URML, there are two curves that seem to be different from the others (See Fig. 7.38). In fact these two participants did not study and have no IT background. The two curves are however not very similar. We can't explain this difference by any of the data gathered. Similarly, we can see a curve on the error rate chart of KAOS (Fig. 7.39), that is clearly different from the others, again a participant with no IT background. On URN, there has also been a non-IT participant, but that curve is not easily seen (Fig. 7.40). The data we have is too limited to conclude anything from this, but suggests that it might be interesting to put more work in finding participants from different domains in future repetitions of this experiment.

)



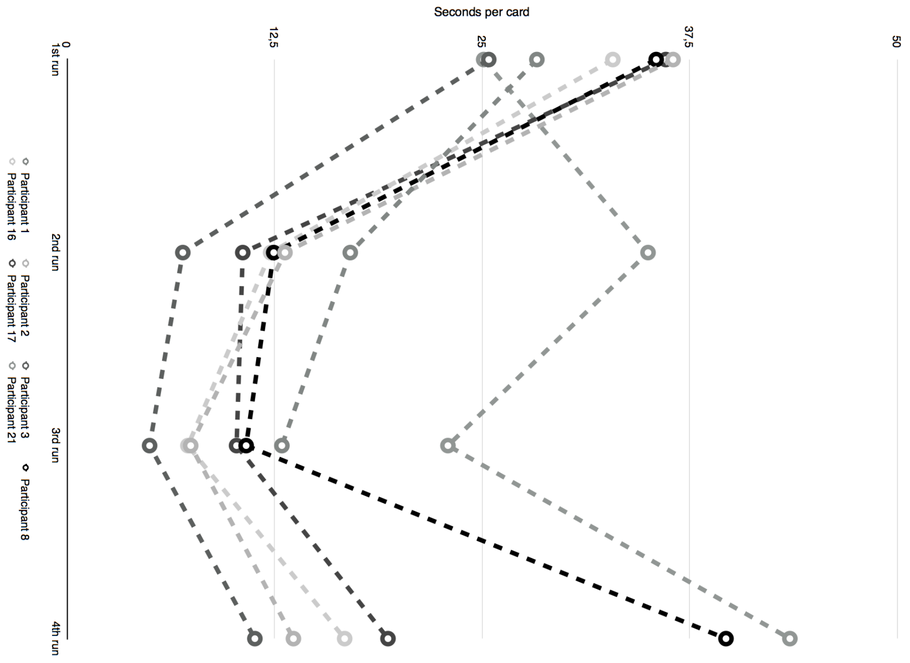


Figure 7.38.: Error rates and seconds per mapping of the individual participants on URML

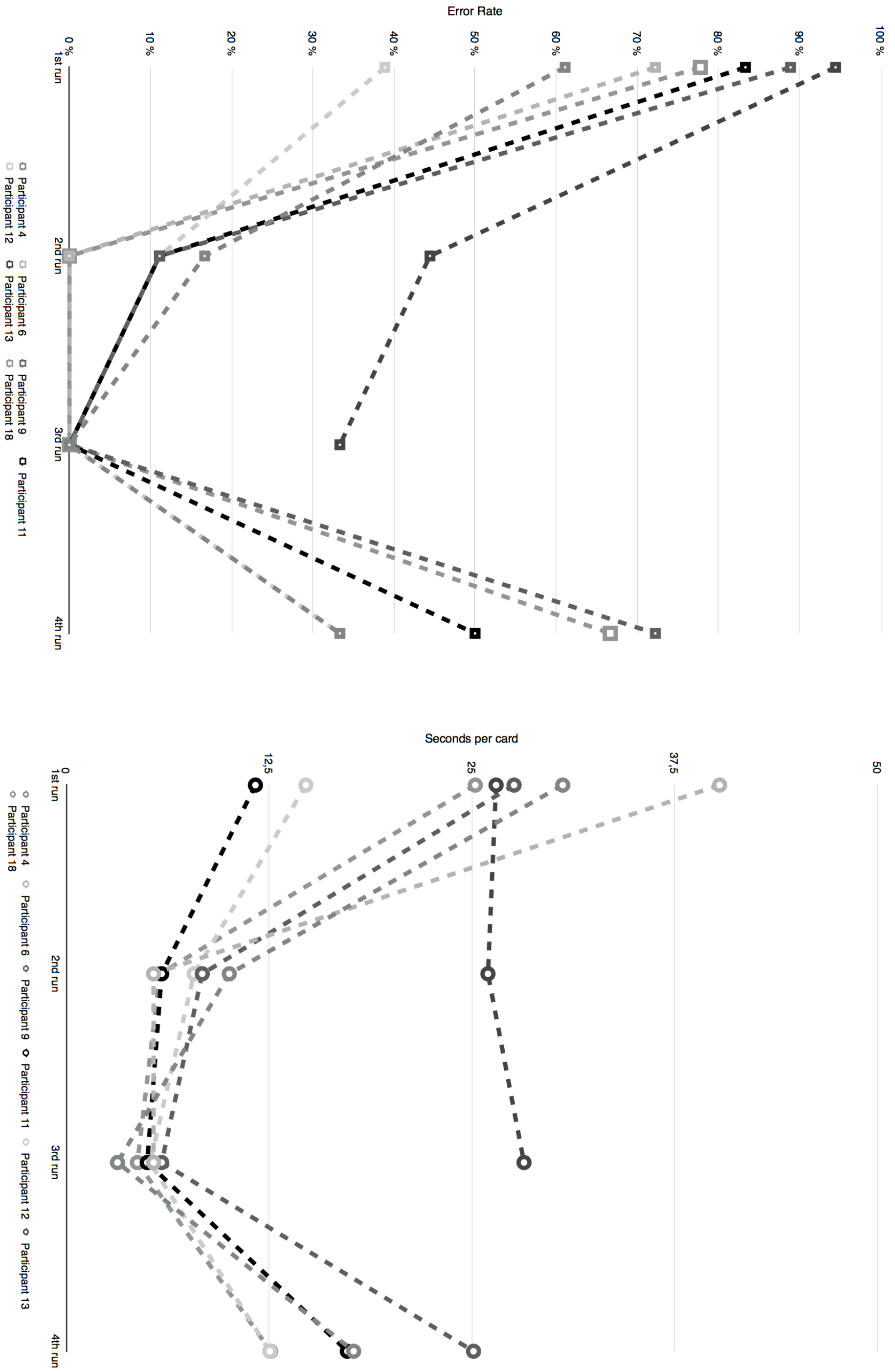
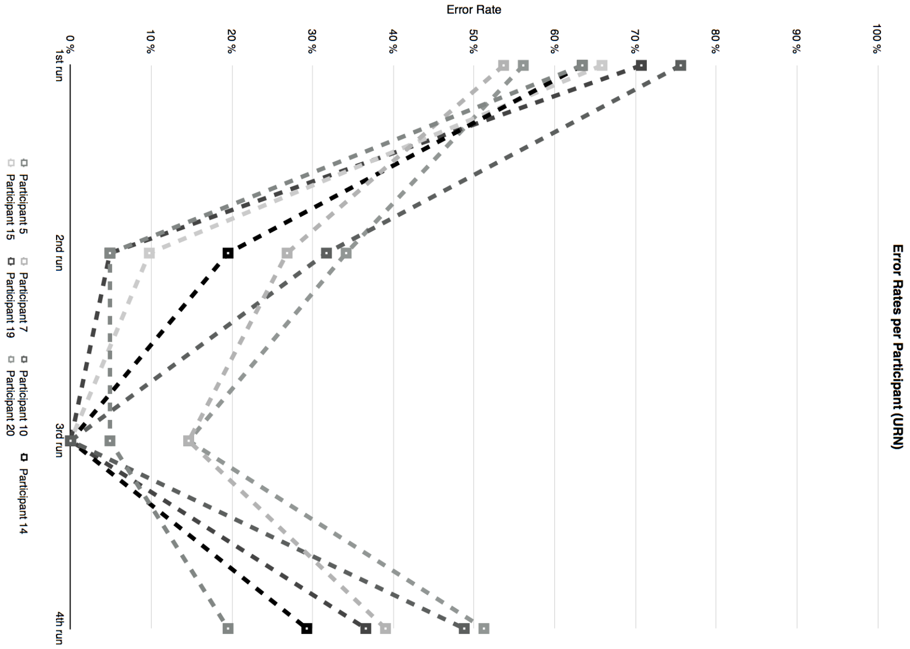


Figure 7.39.: Error rates and seconds per mapping of the individual participants on KAOS



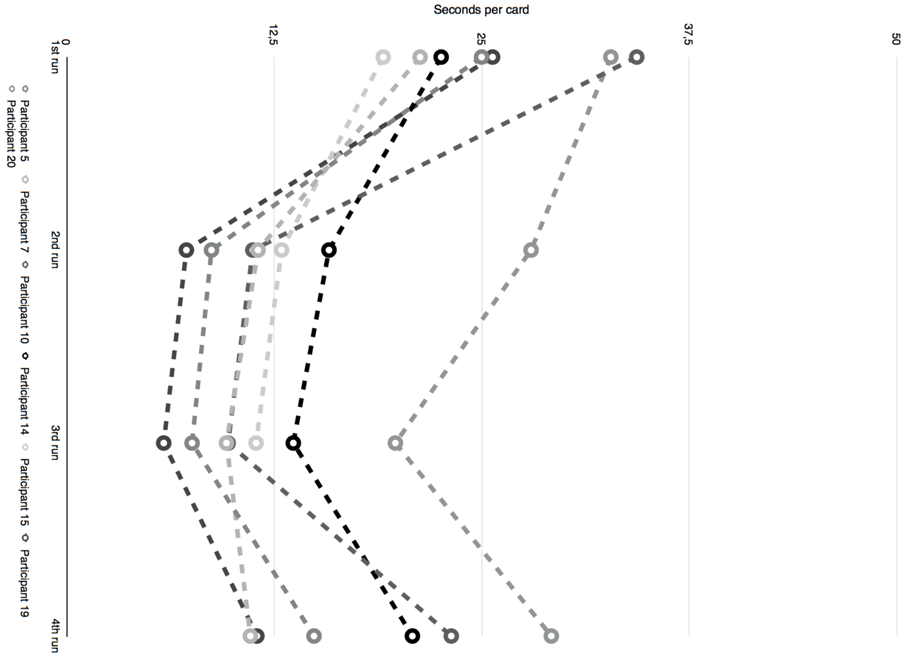


Figure 7.40.: Error rates and seconds per mapping of the individual participants on URN

Beyond the comparison of the languages, we can discuss the performance of the individual symbols. We therefore recorded for each run the erroneous mappings, and then counted how often a particular symbol was part of erroneous mappings[3]. We call this an error on the particular concept. In the following paragraphs, we will use the word concept when not explicitly talking about a specific term or symbol. For the data upon which this summary is based, see appendices A.10.2 and A.10.3.

In the **URML** experiment, the seven participants made 275 errors on 1593 mappings. If these errors were evenly distributed among the concepts, each concept would have had 4.66 errors. The following concept (or groups of concepts) were easy to grasp for the seven participants, as only two or less errors were made on each particular concept in total: *Asset*, *Danger*, *Feature Tree*, *Idea*, *Human Service Provider*, *Customer*, *Business Stakeholder*, and *Usability*, *Efficiency*, and *Maintainability Quality Requirement.*
The concept with the highest total number of errors was *System under Discussion* (thirteen errors), followed by *Procedure* (twelve errors). *Stakeholder*, *Functional Requirement*, *Assessment Sketch*, *Composite Entity Object*, and *Boundary Object* had ten errors each. *System under Discussion, Boundary Object,* and *Atomic Entity Object* were mapped incorrectly by all seven participants in the first run. *Procedure*, *Assessment Sketch*, and *Composite Entity Object* each had six errors on the first run. No concept had more than three errors in the second and more than two errors in the third run. The only concept having more than three errors in the fourth run is *Procedure* (four errors). The fourth run was performed by six participants.
The erroneous mapping most often made was mapping the symbol *Procedure* to the term *Assessment Sketch* (five times). This is followed by three erroneous mappings that were done four times: The symbol *Reliability Quality Requirement* was mapped to the term *Regulatory Quality Requirement*, *Project Execution Quality Requirement* to *Performance Quality Requirement*, and *Functional Suitability Quality Requirement* to *Functional Requirement.*

Figure 7.41 visualizes those erroneous mappings of the URML that occurred at least two times. The labeled boxes on the left side represent the name cards, whereas the labeled boxes on the right represent the symbol cards. The size of the box represents how often the name or symbol was part of an error. The connections between the boxes represent the erroneous mappings, and the thickness of the line represents how often a concrete erroneous mapping occurred. The colors of the lines were chosen arbitrarily, with the goal to make the lines differentiable. Lines starting from the same box on the left have the same color. The data underlying the diagram can be found in section A.10.3.1. For a thorough analysis of the diagram, it should be viewed side-by-side with the mapping counts.

[3] If a symbol is mapped erroneously to a term, the corresponding correct term is inevitably mapped erroneously to another symbol. The count how often a particular term is part of an erroneous mapping will result in the same number how often a symbol is part of an erroneous mapping.

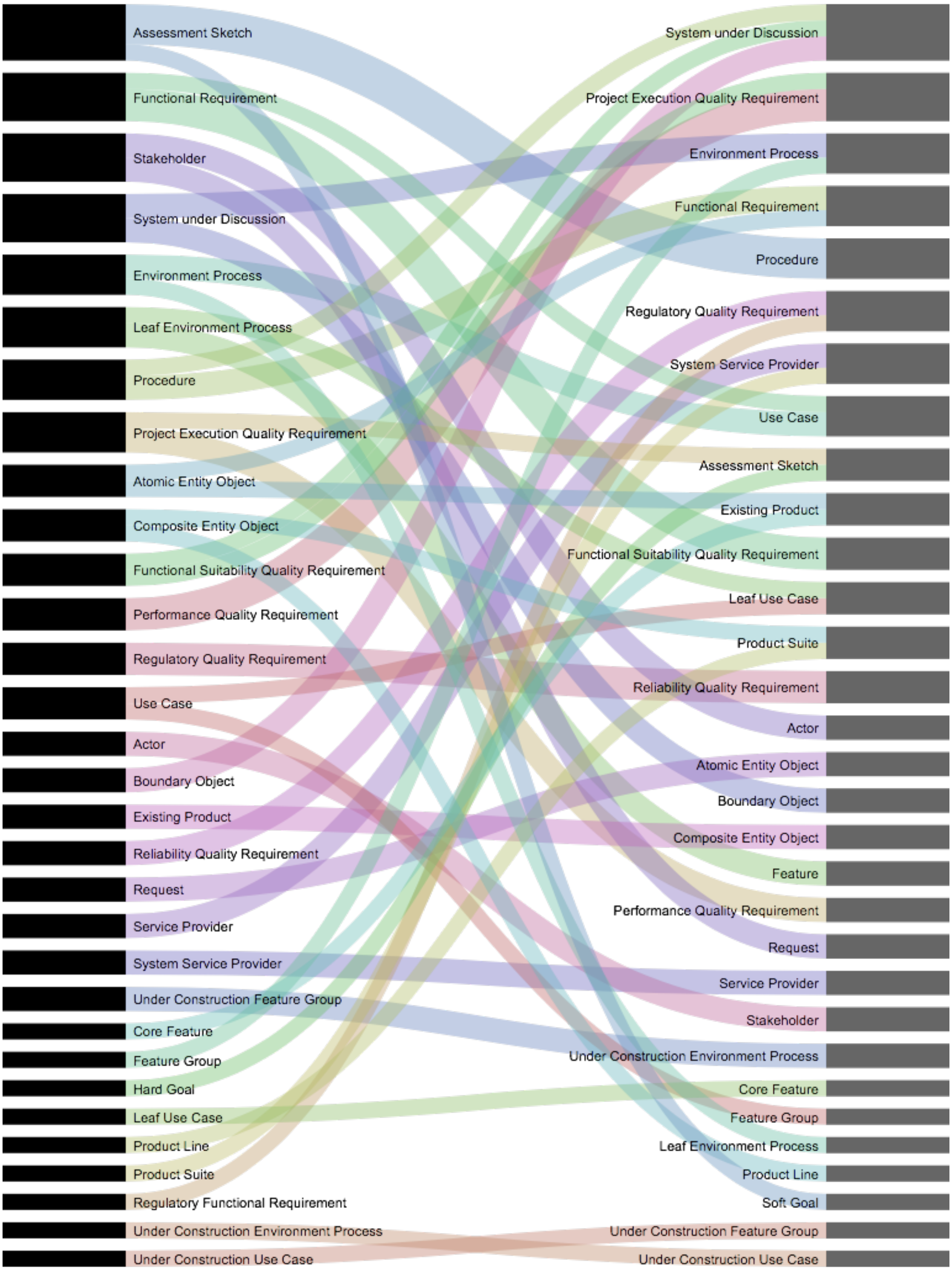


Figure 7.41.: The most often made erroneous mappings in the URML Experiment

In general, the diagram can only be seen as a trend. We assume that if there are strong interactions between concepts already in the experiment with seven participants, they will manifest themselves as strongly in an experiment with more participants. However, erroneous mappings now not in the picture might become stronger. In the following, we discuss only few of the lines in the diagram, to outline how the diagram can be read. For example, we can see a really strong interaction between Assessment Sketch and the Procedure. Of the six times a wrong symbol was put on the Assessment Sketch name card, five times the symbol card was the one of Procedure. It is also visible that the System under Discussion symbol has a problem, but it was not always mapped to the same name card. The number of lines starting from the right System under Discussion box would be even greater if the diagram contained all erroneous mappings.

In the **KAOS** experiment, the seven participants made 162 errors on 468 mappings, which amounts to 9 errors per concept on average. Only one concept was mapped without any problems: There was only one error on *Timeline*. All other concepts had five or more errors in total.
The concept with the highest number of total errors is *Goal*, which was mapped erroneously fifteen times. It is followed by *Requirement* and *Object* (thirteen errors), and *Soft Goal* (twelve errors). In the first run, five concepts were done wrong by all participants (i.e. seven errors): *Goal*, *Requirement*, *Soft Goal*, *Entity*, and *Domain Property*. *Object*, *Expectation*, and *Event* had six errors on this run. No concept had more than two errors in the second run and more than one error in the third run. In KAOS, two participants did not perform the fourth run, so the highest possible error count is five for this run. *Goal*, *Requirement*, and *Object* have this error count. *Expectation* and *Entity* have four errors; *Operation*, *Soft Goal*, and *State Machine* have three errors.
The erroneous mapping most often made was mapping the symbol *Requirement* to the term *Goal* (eight times), the second most was mapping the symbol *Entity* to the term *Object* (seven times). This is followed by mappings *Environment Agent* to *Software-To-Be Agent*, *Expectation* to *Requirement*, and *Software-To-Be Agent* to *Environment Agent* (six times each). The *Goal* symbol was mapped five times to the term *Soft Goal*.
The graphical visualization of the erroneous mappings on KAOS can be visualized in one graph when omitting the mappings that happened only once (Fig. 7.42). We can see that one pair of concepts is separated from the rest of the graph: Environment Agent and Software-To-Be Agent are most of the time confused with each other (six out of eight errors). Similarly, Final State and Initial State were confused with each other in four out of six erroneous mappings. Model Annotation was confused surprisingly often with Soft Goal. Entity was put very often (seven errors) on Object, which was conversely also sometimes mapped to Entity (four errors), but also to the same amount to Goal. Interesting is the loop in the graph, consisting of Soft Goal, which was mapped four times to Expectation, which was mapped six times to Requirement, which was mapped eight times to Goal, that was in turn mapped five times to Soft Goal. Operation participates in many errors, but we can see no tendency with which concept those errors occur most often.

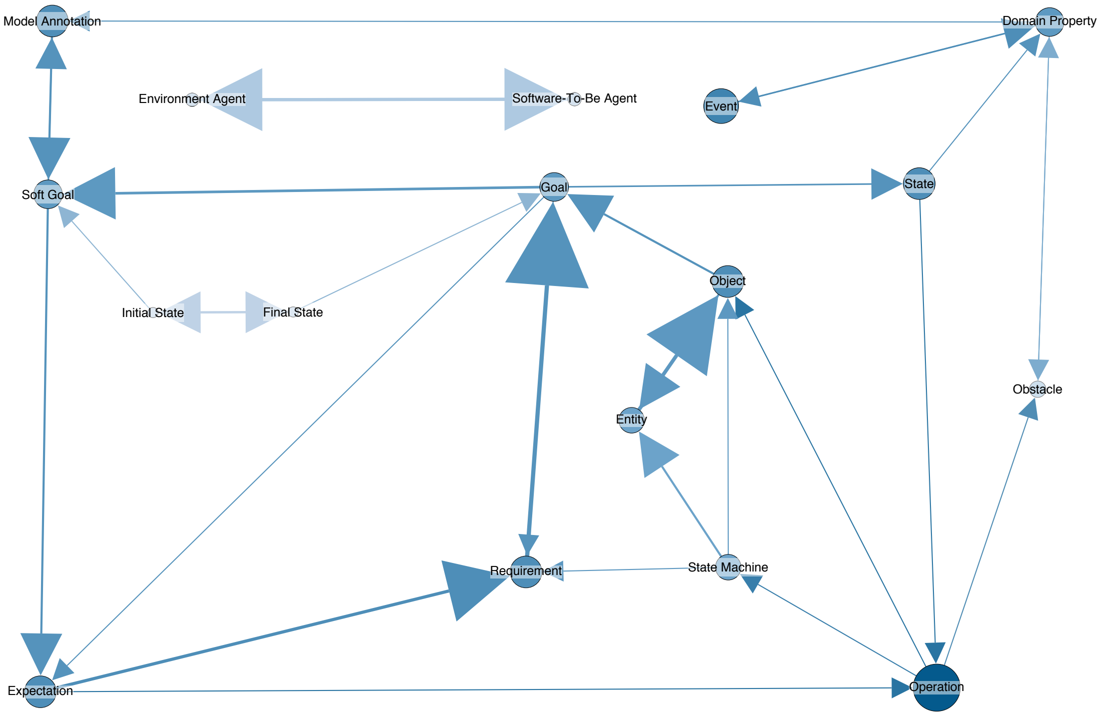


Figure 7.42.: The erroneous mappings of KAOS concepts, edges with weight one omitted.

In the **URN** experiment, the seven participants made 344 errors on 1107 mappings, amounting to an average of 8.39 errors per concept. The concepts with the least errors were *Satisfaction Level "conflict"*, -*"denied"*, and -*"weakly denied"* (no error), followed by *Satisfaction Level "weakly satisfied"*, -*"satisfied"*, and *Empty Point* (one error), and *Start Point* (two errors). The other concepts had four or more errors in total.
The concept with the highest number of total errors is *Team* (sixteen errors), followed by *Protected Team* and *Belief* (fifteen errors). *Task*, *Indicator*, *Object*, and *Protected Object* had fourteen errors, *Agent*, *Protected Agent*, and *Failure Point* had thirteen errors. For URN, there is large number of concepts that all seven participants had wrong in the first run: *Team*, *Belief*, *Task*, *Indicator*, *Object*, *Protected Object*, *Failure Point*, *Abort Start Point*, *Protected Process*, *Process*, and *Actor (collapsed)*. Six out of seven participants had *Protected Team*, *Agent*, *Protected Agent*, *Resource*, *Waiting Place*, and *Actor (expanded)* wrong in the first run. In the second run, four out of seven participants had *Team* and *Protected Team* wrong. All other concepts had three errors or less, the larger part two errors or less. Most of the concepts had no errors in the third run, some had one error. Six participants completed the fourth run, so the highest possible error count is six here. Five concepts had five errors here: *Team*, *Protected Team*, *Object*, *Protected Object*, and *Failure Point*. Six concepts had four errors: *Belief*, *Indicator*, *Agent*, *Protected Agent*, *Abort Start Point*, and *Failure Start Point*.
The erroneous mappings most often made were mapping the symbol *Abort Start Point*

to the term *Failure Start Point* and mapping the symbol *Failure Start Point* to the term *Failure Point* (seven times). *Static Stub* was mapped to *Dynamic Stub* six times. Five times wrong were the mappings *Synchronizing Stub* to *Static Stub*, *Satisfaction Level "unknown"* to *Satisfaction Level "unspecified"* and vice versa, *Protected Agent* to *Protected Team*, and *Protected Team* to *Protected Object*.

7.2.4. Interpretation of Results

The data gathered from the experiment suggests that URML symbols are easier to recognize than those of KAOS or URN. URML had the lowest initial error rate, which shows that it has more icons that are semantically immediate, i.e. the participants could intuitively create a correct mapping on the first run. Also URML had the best retention rate, the error rate on the fourth run was even lower than on the first. This is due to the high semantic translucency of many symbols. The ones that are semantically immediate were usually done right in the first run, but the translucent ones can be memorized after the sample solution has been presented once.

Apart from the comparison of the languages, the experiment gave indications about flaws in the notation design of the three languages. Even though we would have had stronger indications with more participants, we assume that it is likely that symbols that had many errors in our seven-participant-experiment, would also have in an experiment with more participants. The symbols and terms discussed in the following paragraphs are good candidates for future improvement.

In the URML experiment, we have seen that System under Discussion, Procedure, and Functional Requirement were among the concepts with the most errors. This is because the symbols are semantically opaque. But this can also be traced back to their symbols having a non-optimal visual distance, and thus reduced perceptual discriminability. All three, while having distinct semantics, share the cog wheel as part of the symbol. This confused some participants in runs two to four. System under Discussion had another problem: It was confused with Boundary Object. One participant reported that she chose that mapping because the System under Discussion symbol had the "boundary" she was actually searching for when looking for a symbol matching the term "Boundary Object". The symbol of Boundary Object, in turn, has nothing that could be interpreted as a boundary. Thus its symbol has a low semantic translucency.

Functional Requirement had a strong contribution from Functional Suitability Requirement, which we explain as follows. Once the participants had identified the symbolic pattern that diamonds with letters inside stand for certain quality requirements, their tendency was to find something that started with the letter and ended with "requirement", overlooking that "Functional Requirement" was missing "Quality". Thus having the letter "F" inside the diamond symbol does not support the perceptual discriminability of Functional Suitability Requirement enough. A similar problem arose for the Performance Quality Requirement symbol that was often placed on the Project Execution Quality Requirement term. That the Project Execution Quality Requirement has no letter inside, but another symbol, has contributed to the problem.

The Request symbol is unnecessarily close to the one of Stakeholder, due to the shar-

ing of the symbolical person. Therefore, the Request symbol had been placed on the Stakeholder term several times. Due to the exclamation mark in the stakeholder symbol, Stakeholder was not placed on Request to the same amount. Stakeholder was however often confounded with Actor and vice versa. The Atomic Entity Object symbol was often misinterpreted as “message”, which is why it was often interpreted as Request, adding to the confusing regarding Request. Regarding the confusing around Atomic Entity Object: It has no common base symbol with Composite Entity Object, usually a pattern we followed for the design of other families of symbols. The participants had the biggest problem with the term Assessment Sketch. Its symbol focuses on assessment of goals (a scale weighing a goal) but as the participants were not trained in URML (due to the nature of the experiment) nobody could know the relationship between Assessment Sketch and Hard Goal that is present in the meta-model. Most participants were confused by the term itself and could not make sense of it. Some participants did not even recognize a scale in the symbol. Many searched for a symbol having something “sketchy”, which might explain that the Procedure was chosen that often, because its symbol could be read as “something abstract within a book”.
The duality of Use Case and Environment Process symbols led to much confusion. All three variants of them were confused with each other, and the Leaf variant was often confused with the base variant. While the relationship between the terms Use Case and Environment Process is not immediately clear for the novice user, the symbols are usually seen as belonging to a family. Usually then the participants did a 50/50 guess whether to put the oval symbol on a *Use Case or a *Environment Process card. However, as most people were more likely to mentally connect the computer symbol on Use Case with something process-like, most errors were due to the mapping of Use Case symbol to the Environment Process term. The Environment Process symbol in turn was very often put on System Under Discussion, because of the “shaking hands” part of the symbol. We think not only the semantic opacity of the family of process symbols is a reason for the many errors. Also the mis-alignment of the terms Use Case and Environment Process has contributed to the confusion.

Another class of errors in the URML experiment is related to adornments that could be put on several different symbols, as Under Construction (applies to Feature Group, Use Case, and Environment Process), Regulatory (applies to Functional Requirement, Quality Requirement and all its subclasses), and Leaf (applies to Use Case and Environment Process). Under Construction and Regulatory were often easily identified, but then the main icon lead to confusion, sometimes leading to a correction of an already correctly mapped symbol. Leaf is only represented by a dot, which was not easily brought together with “leaf”, therefore Leaf Use Case or Leaf Environment Process have been confounded with their non-leaf counterparts.

8. Conclusion and Future Work

In this dissertation, we have presented the Unified Requirements Modeling Language (URML), a graphical requirements modeling language. URML represents one concrete proposal for a *unified requirements modeling language* for early requirements engineering. To achieve this, we have integrated abstractions from different competing requirements modeling approaches in one meta-model. Among the abstractions integrated are Danger, Feature, Goal, Stakeholder, Actor, Use Case, System, and Requirement. We followed a *holistic approach*: Instead of preferring one abstraction over the other, we have made an attempt to integrate all the abstractions needed for early requirements modeling, by integrating abstractions from languages that have already been scientifically validated. This has been motivated by the problem that a requirements analyst currently has to choose between competing approaches for graphical requirements modeling, and that we already have observed a trend towards *convergence* of modeling concepts in other approaches. URML contributes to the discussion how a unified requirements modeling language could look like. On the basis of URML, future research can investigate benefits of the unified model.

As a second contribution, we have outlined a framework to specify *requirements elicitation strategies*. This was inspired by earlier work done in KAOS. The higher the number of abstractions in the language's meta-model, the greater the *flexibility in switching between elicitation strategies* . With URML, the requirements analyst can choose between goal-driven, feature-driven, use case-driven, or stakeholder-driven strategies.

The third contribution focuses on the visual notation of URML. First, we have presented a novel style of visual notation that uses icons as main constituent. Second, the notation of URML has been designed to address Moody's "Physics of Notations". We conducted an experiment to evaluate and compare the semantic transparency of three languages. The results of our experiment suggest that the symbols of URML are *easier to learn and remember.* URML, in spite of having a much larger set of symbols than URN and KAOS, had lower average error rates than the other two languages. More work needs to be done in this direction, as the intuitiveness of the symbols is only a part of providing a modeling language that is easy to learn.

Third, URML's visual notation has been specified as a concrete syntax meta-model with a concrete-to-abstract syntax mapping. To the best of our knowledge, only URN provides a comparable concrete syntax meta-model.

Though the comparability of graphical requirements modeling languages has not been a main focus of this dissertation, we have contributed to this topic by comparing URML, KAOS, and URN. Besides presenting URML, we have also analyzed the abstract syntax meta-models and the notations of KAOS and URN. We suggest more work to be done regarding the comparability of graphical modeling languages. Concepts and frameworks should be developed to provide a scientific basis on which the utility and cognitive effectiveness of languages can be evaluated and compared. This will allow for future languages being designed in a less ad-hoc style and can lead to an improvement of existing languages.

We have identified four different areas in which future improvements can take place. The first area is about unification, which entails incorporating new abstractions to or modifying the current meta-model, to make URML even more a unified requirements modeling language. The second area, closely connected to the first, is about notation. Some of Moody's principles weren't fully addressed within this dissertation or still can be improved. Examples are cognitive fit and complexity management. Regarding both areas, additional tool support may be required. The fourth area is about enhanced tool support for URML. The third area is about empirical evaluation of requirements modeling languages, and how existing RML's can be comprehensively compared to each other.

With respect to the first area, URML can be improved by integrating ISO standards on quality requirements and constraints (ISO 2010; ISO 2005). An extension of the requirements taxonomy would allow for higher precision in requirements diagrams. As a byproduct, this would resolve a semantic issue of URML: URML currently only differentiates between functional and quality requirements. Figure 8.1 provides a sketch of how an extension of URML could look like. Especially quality in use would be easier to express, when it could be highlighted which `QualityInUseRequirements` affect the participation of certain `Actors` in certain `UseCases`.

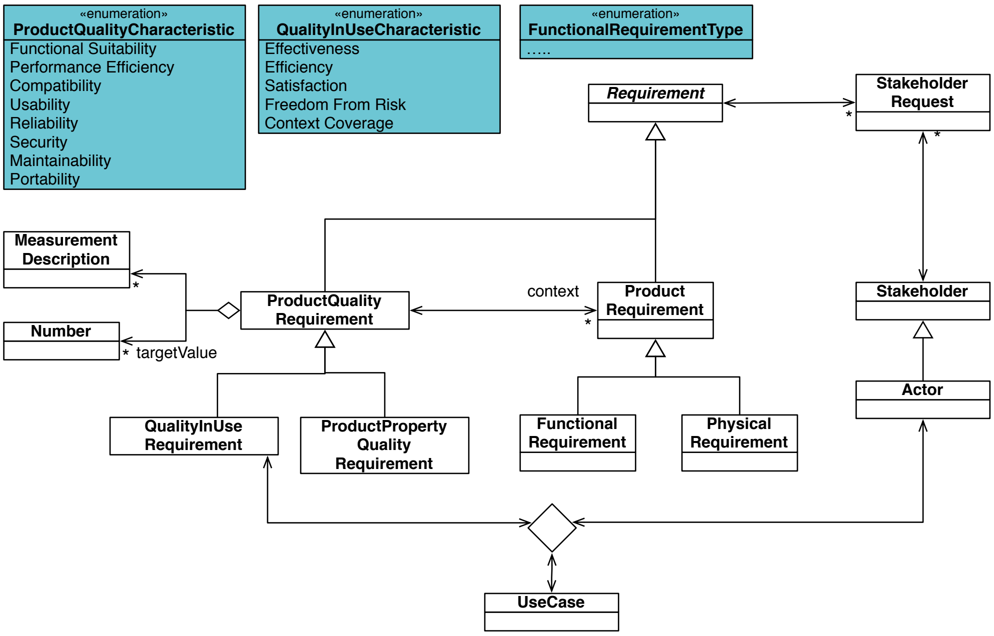


Figure 8.1.: A draft extension of URML's requirements model

There are requirements approaches with significant impact in academia or industry, that have not been incorporated into URML. For example in agile software development methodologies, analysts are working with user stories ("User Stories" 2016) and personas (Cooper 1999). On the list of concepts to integrate in URML, these two abstractions were given a lower priority, as they are semantically close to use case and actor. For the same reason, scenarios (Carroll 2000) were not included, even though they are an important topic in requirements engineering (Potts 1995; Rolland and Salinesi 2009). A thorough discussion of whether the three abstractions should be part of a unified requirements modeling language is needed. URML is currently not explicitly aligned with the reference model described by Gunter et al. (Gunter et al. 2000). A potential alignment of URML with Jackson's Problem Frames approach (Jackson 2001) would increase URML's support for describing requirements patterns.

Regarding the second area, to improve the *cognitive fit* of URML, a complementary notation should be added to the one described in this dissertation, to allow for *drawability* of URML diagrams. The visual style of the current notation that largely relies on icons is hard to draw. Cognitive fit can also be enhanced by defining a subset of the current notation to be used by novices. Certain symbol adornments could be temporarily excluded from the notation to ease the learning process.
New mechanisms for *complexity management* should be investigated and added to URML. Existing approaches like the view concept of ADORA should be evaluated. The abstract syntax meta-model should be extended to allow *relationship compositions:* Compositions would allow to visually shorten the path between two relationships. For example when modeling safety with URML, two relationships are considered, Harm and AssetOwnership, to see wether a Stakeholder is affected by a Danger via his Assets. To reduce complexity (e.g. in order to get an overview), the two relationships could be temporarily composed into one.
As the focus of this dissertation has been more on the notation for entity meta-classes, more work on the notation for relationships is needed. New concepts beyond variations of the line pattern or routing style need to be investigated.

As an improvement in terms of tool support, the framework of Chapter 5 should be implemented as an elicitation dialog system that in a visionary state would also be usable by stakeholders.
In certain domains, not all abstractions of a modeling language are necessary to encode the requirements knowledge. Therefore, the tailorability of language meta-models could be investigated. Before creating a model, users can configure which entities of the meta-model should be in- or excluded (See example in Figure 8.2). That leads to some interesting questions, for instance how a tool can adequately support the tailoring, and especially regarding model migration if the configuration of a meta-model is changed during the project. In the example figure, the user has chosen not to model features. Three options for adjusting the meta-model exist: 1. Remove the excluded class and all relationships it participates in, 2. Connect GoalRealization to Requirement. 3. Remove GoalRealization, rename FeatureConstraint and FeatureDescriptionRequirement, and connect them to Goal.

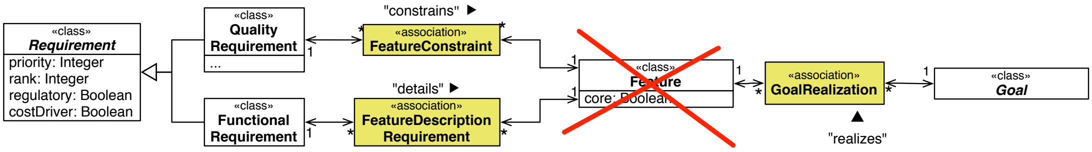


Figure 8.2.: Tailorable meta-model problem: Following each configuration change, how can the meta-model adequately be adjusted?

Regarding fourth area, some *replication* of the conducted experiment is advisable. Our experiment had only twenty-one participants, with the majority having an IT background. Replications would involve more participants, from a broader range of professions. Apart from the profession, the job position of participants could be recorded in the questionnaire, to analyze requirements for cognitive fit.

Further experiments, which are focused on other aspects of the notation than the entity symbols, should be conducted. In URN, KAOS, or ADORA, certain spatial arrangements on a diagram have certain semantics. Examples are the dotted-line oval of an instance of `Actor (expanded)` encompassing several instances of `Goal` in URN, or the `State Machine` symbol containing multiple symbols of `State` in KAOS. As URML does not have such features, the conducted experiment was not suitable to compare that aspect.

The *relationship notation* was not tested in the conducted experiment. We suggest that a dedicated experiment should be constructed, to account for the peculiarities of relationship notations. For example, an experiment could measure relationship comprehension on diagrams, and whether languages with a strong focus on textual differentiation of relationship types will perform worse than relationship notations where labels on relationship lines are used for dual coding.

For evaluating the cognitive effectiveness of a language, the conducted experiment is not sufficient. Further experiments testing *diagram comprehension*, can provide insights on the benefit of complexity management and cognitive integration mechanisms of a notation.

Finally, we envision *real case studies* to provide valuable insight into the applicability of URML. Companies willing to participate in such studies need to be found, which allow for presentation of the created models. That might be a challenging task as companies are usually not willing to disclose proprietary information. In addition there may be reluctancy to being connected with a requirements model that may be considered suboptimal in the worst case. Therefore, small first steps into that direction could be done with student teams at university, who have to deliver a real product. That approach might facilitate a comparative study as well, in which different requirements modeling languages are compared. However such a study would only allow for the analysis of the usage of a language by untrained users (i.e. that are no requirements experts yet).

In order to have real requirements experts create a requirements model for a real problem, we envision an open call for tender, which would request the creation of a requirements model for a real product. The submissions would be judged by a mixed committee of requirements researchers and industry experts, and the company with the winning bid would get the contract for developing the product. Data gathered during the downstream development process could add further insights, e.g. which requirements were discovered during design, development, or testing. A series of such projects might finally provide a fair comparison of available requirements modeling languages.

A. Appendix

The main purpose of this appendix is to to present a detailed specification of the URML as it was implemented in version 1.0.5 of our reference implementation. Apart from showing one diagram per package of the URML, each element of the meta-model is described. The structure of the such a description is described first in section A.3.

A.1. URML Abstract Syntax Meta-Model

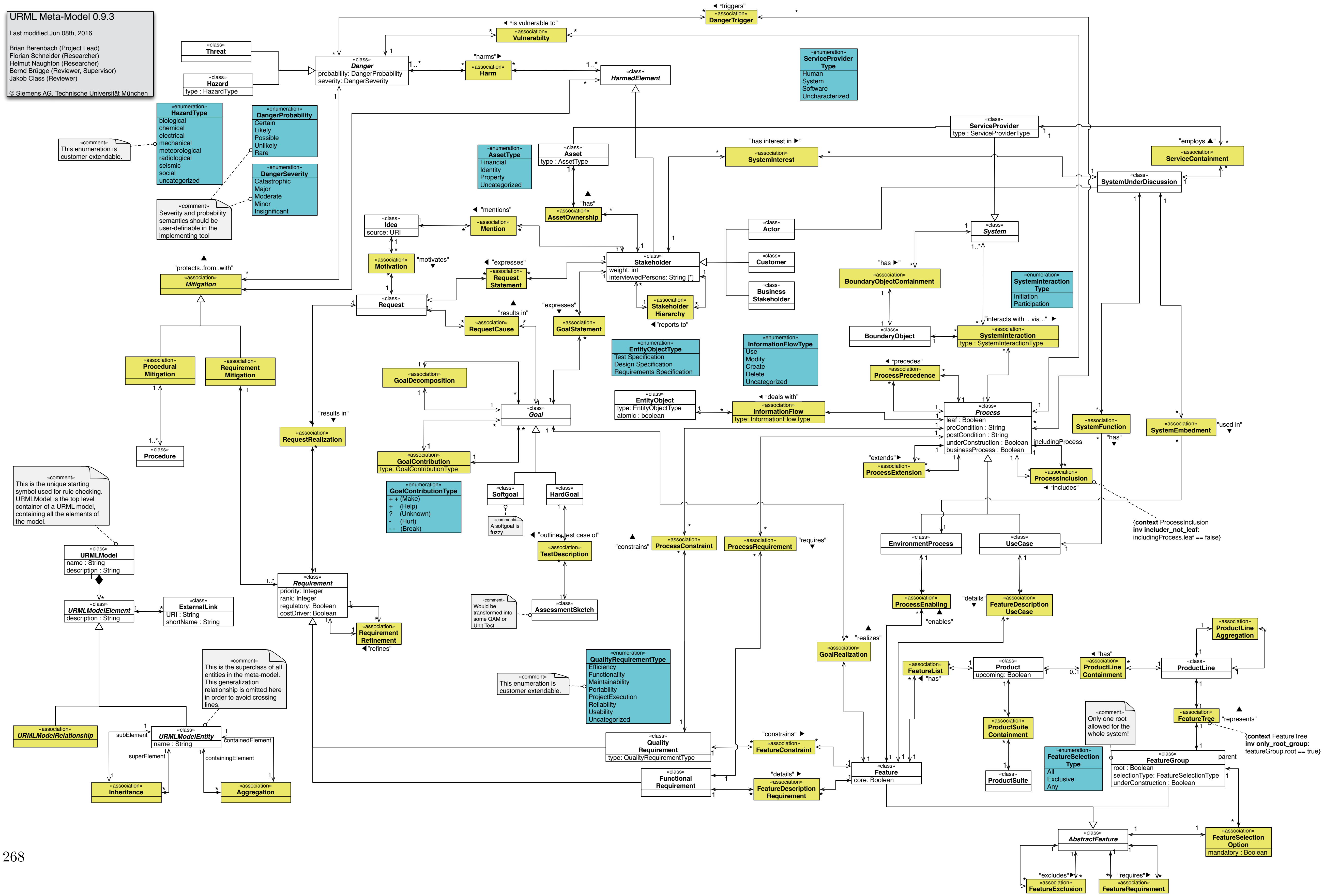

URML Meta-Model 0.9.3
Last modified Jun 08th, 2016
Brian Berenbach (Project Lead)
Florian Schneider (Researcher)
Helmut Naughton (Researcher)
Bernd Brügge (Reviewer, Supervisor)
Jakob Class (Reviewer)
© Siemens AG, Technische Universität München
«class» Threat
«class» Hazard
type : HazardType
«class» Danger
probability: DangerProbability
severity: DangerSeverity
"harms"
«association» Harm
«class» HarmedElement
"is vulnerable to"
«association» Vulnerabilty
"triggers"
«association» DangerTrigger
«enumeration» ServiceProvider Type
Human
System
Software
Uncharacterized
«enumeration» HazardType
biological
chemical
electrical
mechanical
meteorological
radiological
seismic
social
uncategorized
«comment» This enumeration is customer extendable.
«enumeration» DangerProbability
Certain
Likely
Possible
Unlikely
Rare
«enumeration» DangerSeverity
Catastrophic
Major
Moderate
Minor
Insignificant
«comment» Severity and probability semantics should be user-definable in the implementing tool
«enumeration» AssetType
Financial
Identity
Property
Uncategorized
«class» Asset
type : AssetType
"has"
«association» AssetOwnership
"has interest in"
«association» SystemInterest
«class» ServiceProvider
type : ServiceProviderType
"employs"
«association» ServiceContainment
«class» SystemUnderDiscussion
"mentions"
«association» Mention
«class» Idea
source: URI
«association» Motivation
"motivates"
"protects..from..with"
«association» Mitigation
"expresses"
«association» Request Statement
«class» Stakeholder
weight: int
interviewedPersons: String [*]
«association» Stakeholder Hierarchy
"reports to"
«class» Actor
«class» Customer
«class» Business Stakeholder
«class» System
"has"
«association» BoundaryObjectContainment
«enumeration» SystemInteraction Type
Initiation
Participation
«class» BoundaryObject
"interacts with .. via .."
«association» SystemInteraction
type : SystemInteractionType
«class» Request
"results in"
«association» RequestCause
"expresses"
«association» GoalStatement
«enumeration» EntityObjectType
Test Specification
Design Specification
Requirements Specification
«enumeration» InformationFlowType
Use
Modify
Create
Delete
Uncategorized
"precedes"
«association» ProcessPrecedence
«association» Procedural Mitigation
«association» Requirement Mitigation
«association» GoalDecomposition
«class» Goal
«class» EntityObject
type: EntityObjectType
atomic : boolean
"deals with"
«association» InformationFlow
type: InformationFlowType
«class» Process
leaf : Boolean
preCondition : String
postCondition : String
underConstruction : Boolean
businessProcess : Boolean
includingProcess
«association» SystemFunction
"has"
«association» SystemEmbedment
"used in"
«class» Procedure
"results in"
«association» RequestRealization
«association» GoalContribution
type: GoalContributionType
«enumeration» GoalContributionType
+ + (Make)
+ (Help)
? (Unknown)
- (Hurt)
- - (Break)
«class» Softgoal
«comment» A softgoal is fuzzy.
«class» HardGoal
"outlines test case of"
«association» TestDescription
"constrains"
«association» ProcessConstraint
«association» ProcessRequirement
"requires"
"extends"
«association» ProcessExtension
«association» ProcessInclusion
"includes"
{context ProcessInclusion inv includer_not_leaf: includingProcess.leaf == false}
«class» EnvironmentProcess
«class» UseCase
«comment» This is the unique starting symbol used for rule checking. URMLModel is the top level container of a URML model, containing all the elements of the model.
«class» URMLModel
name : String
description : String
«class» URMLModelElement
description : String
«class» ExternalLink
URI : String
shortName : String
«class» Requirement
priority: Integer
rank: Integer
regulatory: Boolean
costDriver: Boolean
«association» Requirement Refinement
"refines"
«class» AssessmentSketch
«comment» Would be transformed into some QAM or Unit Test
«association» ProcessEnabling
"enables"
"details"
«association» FeatureDescription UseCase
«association» ProductLine Aggregation
"realizes"
«association» GoalRealization
«association» FeatureList
"has"
«class» Product
upcoming: Boolean
"has"
«association» ProductLine Containment
«class» ProductLine
«enumeration» QualityRequirementType
Efficiency
Functionality
Maintainability
Portability
ProjectExecution
Reliability
Usability
Uncategorized
«comment» This enumeration is customer extendable.
«comment» This is the superclass of all entities in the meta-model. This generalization relationship is omitted here in order to avoid crossing lines.
«association» URMLModelRelationship
«class» URMLModelEntity
name : String
subElement
superElement
containedElement
containingElement
«association» Inheritance
«association» Aggregation
«class» Quality Requirement
type: QualityRequirementType
«class» Functional Requirement
"constrains"
«association» FeatureConstraint
"details"
«association» FeatureDescription Requirement
«class» Feature
core: Boolean
«association» ProductSuite Containment
«class» ProductSuite
«comment» Only one root allowed for the whole system!
«enumeration» FeatureSelection Type
All
Exclusive
Any
«association» FeatureTree
"represents"
{context FeatureTree inv only_root_group: featureGroup.root == true}
«class» FeatureGroup
root : Boolean
selectionType: FeatureSelectionType
underConstruction : Boolean
parent
«association» FeatureSelection Option
mandatory : Boolean
«class» AbstractFeature
"excludes"
«association» FeatureExclusion
"requires"
«association» FeatureRequirement
1
*
1..*
0..1

Figure A.1.: URML Abstract Syntax Meta-Model

A.2. URML Package Interfaces

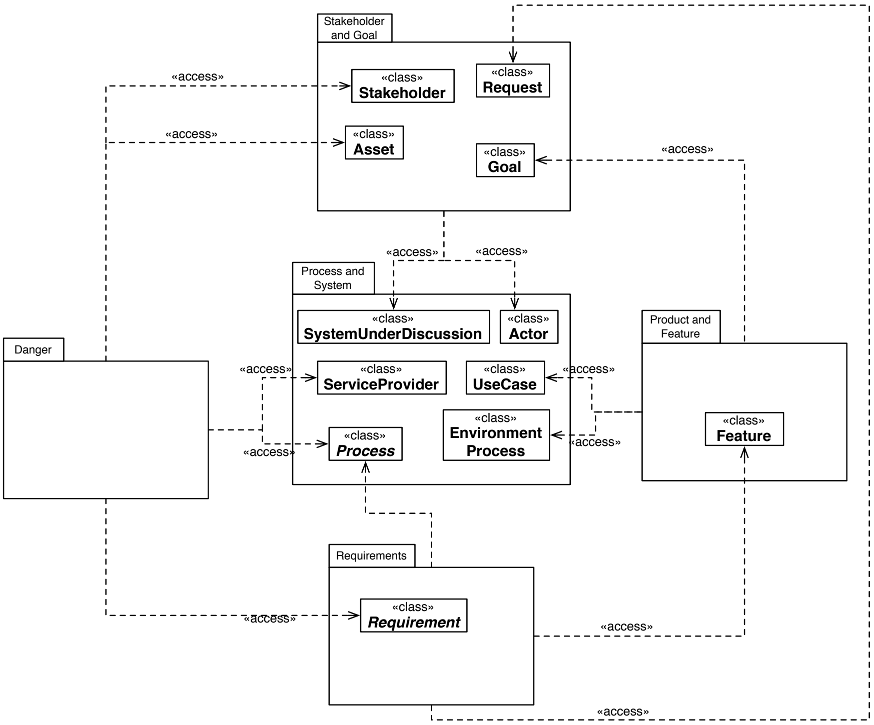


Figure A.2.: URML Package Interfaces

A.3. URML Meta-Meta-Model

This section provides a model-based view on the contents of URML's abstract syntax specification. Through the the diagram in figure A.3, large parts of that section's contents and structure can be explained. Each non-abstract class of the diagram specifies a certain type of subsection of the specification.

The specification describes the URML meta-model and its parts (**NamedModelElement**). All named elements are described with their name, a summarizing sentence, a detailed description, a document giving an example, and an image presenting an excerpt of the graphical meta-model. These named parts of the meta-model are either packages, or elements inside a package (**Package**, **PackagedElement**). Packages are used to group the elements conceptually. Packaged elements have an image showing their notation. They are either enumerations or classifiers (**Enumeration**, **Classifier**). Enumerations have two to many literals. They are used for the categorization of classifiers (**Classifier**). Classifiers can be abstract, have a list of attributes, and a list of constraints. They can be part of a generalization relationship. Classifiers are classes or relationships (**Class**, **Relationship**). Classes participate in relationships, while the incorporating relationship provides role names for the classes.

The packages of the meta-model are described in section 4.2. A description for each enumeration, class, or relationship of the URML is described in section A.4. Each subsection describing an named element provides its own format to list and explain the attributes of the element as shown in figure A.3. For better readability, the sections also provide content regarding the relationships of the meta-types. Enumeration descriptions additionally provide which class they are used to categorize. Classifier descriptions provide a list of generalizations and a list of specializations, if applicable. Class descriptions provide a list of relationships they participate in. Relationship descriptions mention role names and cardinalities of the meta-classes they relate to each other. Descriptions of abstract classifiers have no notation and no example, because the described classifiers are not instantiable and thus cannot appear on a diagram of the URML.

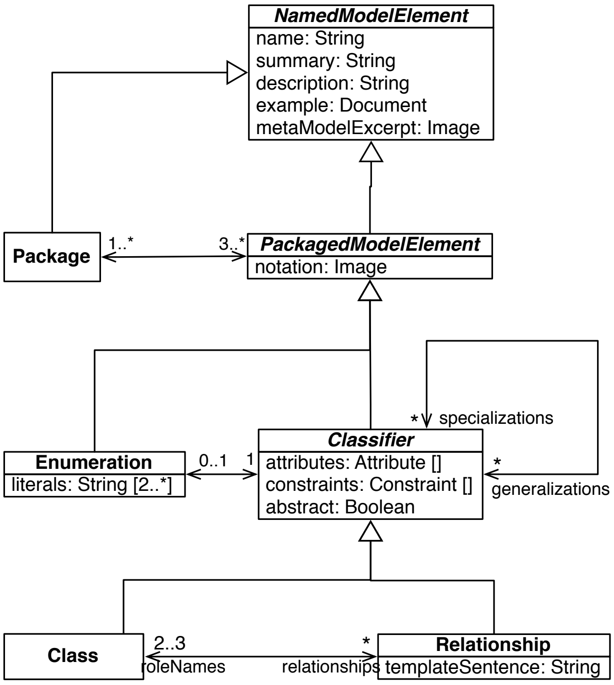


Figure A.3.: A model of the meta-model description

A.4. URML Abstract and Concrete Syntax Per Element

The URML comprises a total of 93 meta-model elements. This section gives a per-element description of classes, relationships and enumerations of the meta-model. The elements are presented in alphabetic, ascending order. Each element is described in its own section. The format of each section is derived from the meta-model as described in section A.3. The name of the element, its metatype, and a specification whether the element is abstract form the section title. This is followed by the element summary, which is one sentence summarizing the function of the element. Further content of the section depends on the metatype of the element.

If the element is a classifier, the summary is followed by a list of the element's super classifiers. This paragraph is named "Generalizations". Then, a detailed description of the element is given. A list of attributes is presented next. Classes also have a listing of the attributes inherited from superclasses. This isn't done for relationships as these are not part of deep inheritance hierarchies.[1]

[1] In the current state of the URML, the only attribute that is inherited by every rela-

For classes, a list of relationships they can participate in comes next, while for relationships a list of role names combined with cardinalities and a specification how relationship instances on a diagram can be "read" (i.e. transformed into prose text) comes next. For both types of classifiers a description of their notation comes next.

If the element is an enumeration, the classifier it is used with is presented. The list of enumeration literals that entail the enumeration comes next, followed by the detailed description. The notation used to reflect different enumeration literals is presented with the notation section of the according classifier, therefore the notation section of an enumeration only references the according classifier's notation section. The specification of an enumeration is concluded by a list of text-based examples.

Conventions In the diagrams showing the abstract-to-concrete syntax mapping of a given element, the role names me and vo reoccur in every diagram. mo is an abbreviation for ModelElement and vo abbreviation for VisualObject. All the display manager classes (suffix -DM) are only showing the constraint compartment (See (OMG 2015c, Section 9.2.4.1)) to define the mapping, i.e. attribute and operation compartments are suppressed. To the best of our knowledge, no special UML notation for inherited constraints exist. Therefore, the diagrams showing the abstract-concrete-syntax-mapping will only show one display manager class, though all of them are part of an inheritance taxonomy (See Figures 4.69, 4.73). The concrete display manager class alsow shows inherited constraints, therefore certain constraints will appear on multiple diagrams. Showing all applicable constraints within one class limits the complexity these diagrams. Furthermore, the relationships of the abstract syntax classes are displayed in the form of properties in the attributes compartment.

tionship is the description attribute of URMLModelElement. The only occasion where the inheritance hierarchy is one level deeper is the taxonomy of mitigation relationships.

A.4.1. AbstractFeature (Abstract Class)

Abstract base class that enables the modeling of feature trees.

Generalizations

- URMLModelEntity

Description Abstract Feature enables the hierarchical structuring of features. It serves as a superclass to Feature and Feature Group. The three form a composite pattern, which allows for modeling the feature tree of a product line. Additionally, the selection of one abstract feature can require the selection or exclusion of other abstract features.

Attributes

- No additional attributes.

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)

A.4.2. Actor (Class)

A system of its own behavior interacting with the system under discussion. Therefore also a stakeholder.

Generalizations

- Stakeholder
- System

Description An actor is a stakeholder that directly interacts with the system under discussion. It can be a living person, or an external entity such as a computer or piece of equipment (e.g. a thermostat). An actor uses boundary objects to initiate processes of the system or to participate in them. An actor is external to the system.

Attributes

- No additional attributes.

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)
- weight (from Stakeholder)

Concrete Syntax and Mapping The notation for Actor is an entity notation. It uses the name attribute of the Actor meta-class, which is used to label the ActorImage denotating instances of Actor. The other attributes of Actor are currently not reflected in the notation.

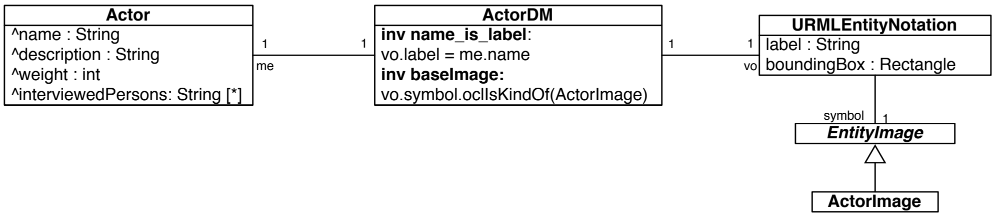


Figure A.4.: Actor Notation

Images ActorDM uses only ActorImage. The ActorImage icon depicts a single person. It was inspired by the stick figure known from UML.

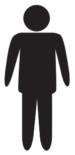


Figure A.5.: ActorImage

A.4.3. Aggregation (Relationship)

Expresses a composite-component relationship for all entities of the URML.

Generalizations

- URMLModelRelationship

Description Aggregation is a relationship of rather generic semantics that is available to every entity of the URML. It offers expression of composition/decomposition semantics between elements of the same kind.

Attributes

- No additional attributes.

Role Names and Cardinalities

composite: URMLModelEntity[1]

component: URMLModelEntity[*]

Template Sentence "URMLModelEntity consists of URMLModelEntit(-y/-ies)"

Concrete Syntax and Mapping The notation for Aggregation is a relationship notation. It has a solid line, no center symbol, an rhombus-style arrowhead at the end of line ending at composite, and no label.

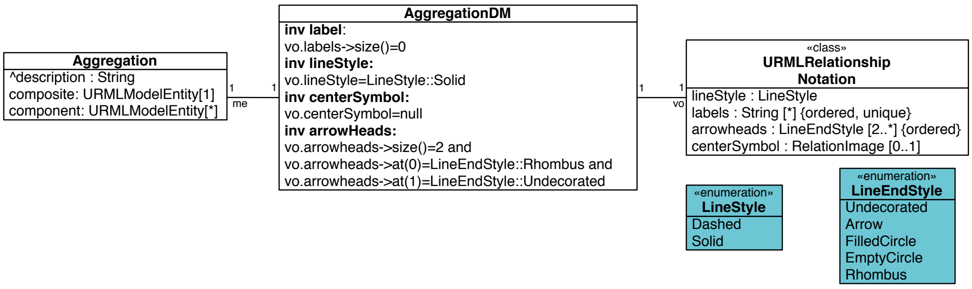

Aggregation
^description : String
composite: URMLModelEntity[1]
component: URMLModelEntity[*]
1
me
1
AggregationDM
inv label:
vo.labels->size()=0
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle::Rhombus and
vo.arrowheads->at(1)=LineEndStyle::Undecorated
1
1
vo
«class»
URMLRelationship Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.6.: Aggregation Notation

Images No special imagery.

A.4.4. AssessmentSketch (Class)

A sketchy description of how the realization of a goal can be tested.

Generalizations

- URMLModelEntity

Description Description of how to measure success in reaching a goal. An assessment sketch is a rather informal element that contains suggestions (in its description attribute) for assessing or testing whether a goal is actually addressed by the system.

Attributes

- No additional attributes.

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)

Concrete Syntax and Mapping The notation for AssessmentSketch is an entity notation. It uses the name attribute of the AssessmentSketch metaclass, which is used to label the AssessmentSketchImage denotating instances of AssessmentSketch. The description attribute of AssessmentSketch is currently not reflected in the notation.

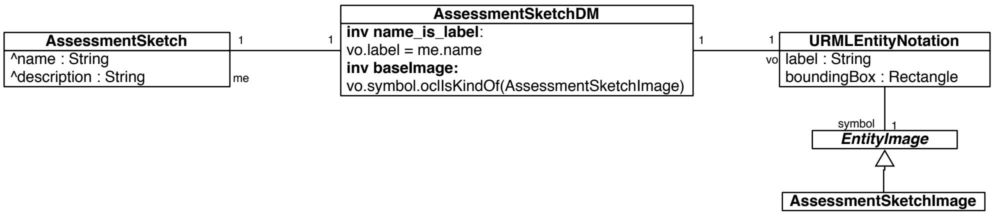


Figure A.7.: AssessmentSketchNotation

Images The icon depicts a goal being weighed on a scale.

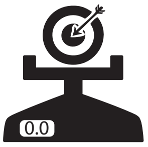


Figure A.8.: AssessmentSketchImage

Known Issues

1. AssessmentSketchImage
   a) the visual element representing a scale might not be recognized as a scale.
   b) The display reading "0.0" in the lower left is irritating and should be represented using something iconographic, for example an iconized analog scale display.

A.4.5. Asset (Class)

Something of value to a stakeholder.

Generalizations

- URMLModelEntity
- HarmedElement

Description An asset is something that is of value to a stakeholder and is owned by the stakeholder. A further specification of the nature of an asset can be given through the asset type. An asset can be harmed by dangers. The nature of the asset, the probability of the danger, and the weight of the stakeholder help in assessing the severity of the danger.

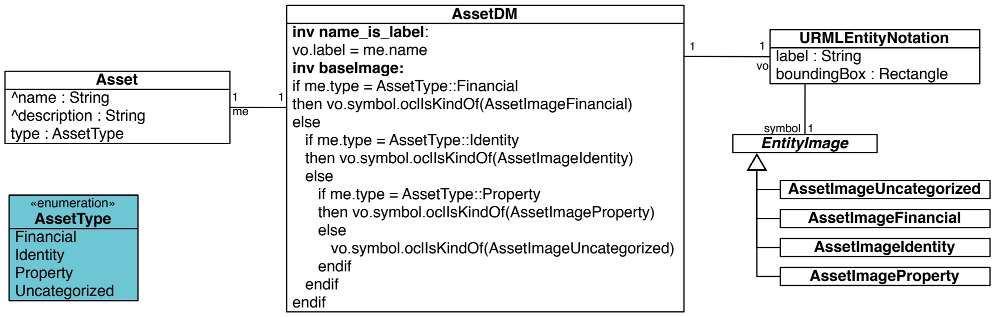
Attributes

- type: AssetType (default value: Uncategorized)

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)

Concrete Syntax and Mapping The notation for Asset is an entity notation. It uses the name attribute of the Asset meta-class to label its instances. The description attribute is currently not reflected in the notation. The symbol varies depending on the value of the type attribute.



```
inv name_is_label:
vo.label = me.name
inv baseImage:
if me.type = AssetType::Financial
then vo.symbol.oclIsKindOf(AssetImageFinancial)
else
  if me.type = AssetType::Identity
  then vo.symbol.oclIsKindOf(AssetImageIdentity)
  else
    if me.type = AssetType::Property
    then vo.symbol.oclIsKindOf(AssetImageProperty)
    else
      vo.symbol.oclIsKindOf(AssetImageUncategorized)
    endif
  endif
endif
```

Figure A.9.: Asset Notation

Images The image for uncategorized assets depicts a money bag. The variations of that image depicting financial, identity, or property assets, fill that bag with content: a dollar sign for financial assets, two office buildings for property assets, and an identity card for identity assets.

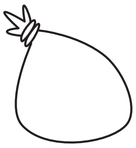


Figure A.10.: AssetImageUncategorized

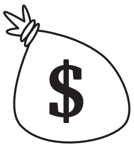


Figure A.11.: AssetImageFinancial

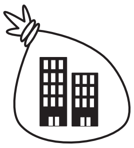


Figure A.12.: AssetImageProperty

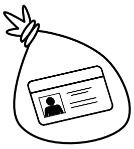


Figure A.13.: AssetImageIdentity

A.4.6. AssetOwnership (Relationship)

Expresses which stakeholder owns which asset.

Generalizations

- URMLModelRelationship

Description This relationship shows which stakeholders own which assets. This is important for understanding goals and requests of the stakeholders, and for the analysis of the impact of dangers.

Attributes

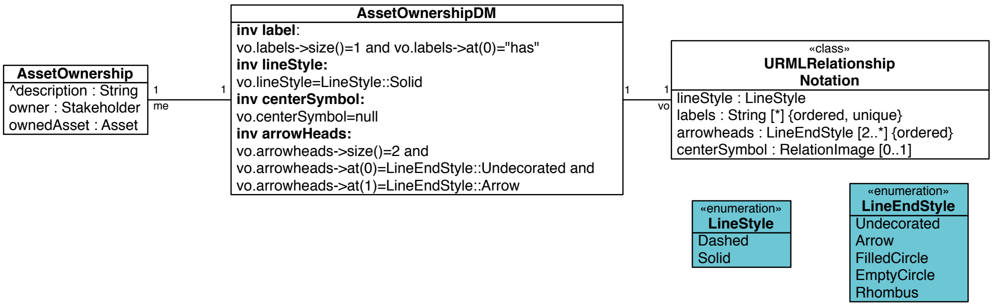
- No additional attributes.

Role Names and Cardinalities

owner:	Stakeholder[1]
ownedAsset:	Asset[1]

Template Sentence “Stakeholder has Asset”

Concrete Syntax and Mapping The notation for AssetOwnership is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at Asset, and the static label text ’has’.



```
inv label:
vo.labels->size()=1 and vo.labels->at(0)="has"
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle::Undecorated and
vo.arrowheads->at(1)=LineEndStyle::Arrow
```

Figure A.14.: AssetOwnership Notation

Images No special imagery.

A.4.7. AssetType (Enumeration)

Enables a distinction between financial, identity, property, and uncategorized assets.

Categorized Class

- Asset

Enumeration Literals

- Financial
- Identity
- Property
- Uncategorized

Description AssetType enumerates the different kinds of assets. A financial asset focuses on assets of monetary value like cash, stock options, or funds. Property assets focuses more on physical assets that might be damaged through a hazard. Identity assets cover personal information about a stakeholder that should be protected. If the distinction is of no importance in a specific context or is unambiguous by the name of the asset, the modeler can leave the asset be uncategorized.

A.4.8. BoundaryObject (Class)

An interface of the system.

Generalizations

- URMLModelEntity

Description The interface between an actor and a system or process. It can for example be a GUI, a panel or a switch. A boundary object is not strictly an input device. While it accepts input by actors it can also possibly communicate or display the responses of the system.

Attributes

- No additional attributes.

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)

Concrete Syntax and Mapping The notation for BoundaryObject is an entity notation. It uses the name attribute of the BoundaryObject meta-class to label its instances. The description attribute is currently not reflected in the notation.

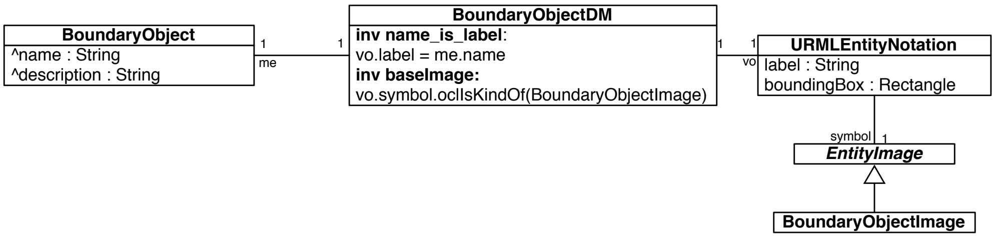


Figure A.15.: BoundaryObject Notation

Images The notation uses one image, BoundaryObjectImage. It depicts a hand, with the index finger touching a big push-button.

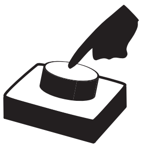


Figure A.16.: BoundaryObjectImage

A.4.9. BoundaryObjectContainment (Relationship)

Shows to which system a boundary object belongs.

Generalizations

- URMLModelRelationship

Description This relationship allows for modeling which boundary objects are exposed by a system. By listing all boundary objects of a system, the modeler can get a summary of the objects through which a system can be used, i.e. the interface of the system. A boundary object may not be part of multiple systems.

Attributes

- No additional attributes

Role Names and Cardinalities

exposingSystem: System[1]

exposedBoundary: BoundaryObject[1]

Template Sentence "System has interface BoundaryObject"

Concrete Syntax and Mapping The notation for BoundaryObjectContainment is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at Asset, and the static label text 'has'.

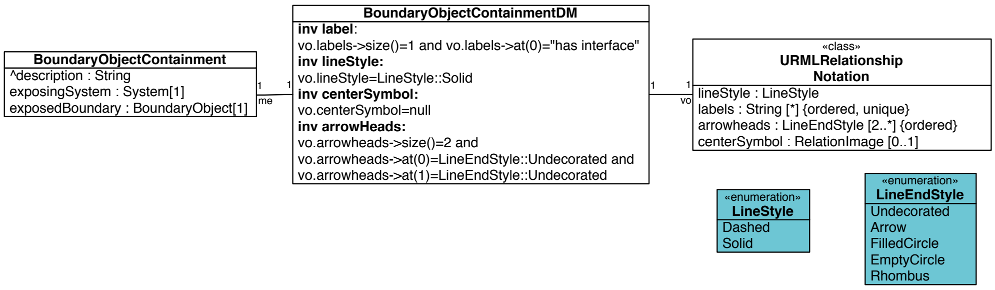

BoundaryObjectContainment
^description : String
exposingSystem : System[1]
exposedBoundary : BoundaryObject[1]
1
1
me
BoundaryObjectContainmentDM
inv label:
vo.labels->size()=1 and vo.labels->at(0)="has interface"
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle::Undecorated and
vo.arrowheads->at(1)=LineEndStyle::Undecorated
1
1
vo
«class»
URMLRelationship
Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.17.: BoundaryObjectContainment Notation

Images No special imagery.

A.4.10. BusinessStakeholder (Class)

Somebody who is primarily interested in the system because he orders it.

Generalizations

- Stakeholder

Description A business stakeholder is a stakeholder other than the customer that is involved with financial aspects of a system or process set. For example, the CEO of a company building rail cars would be a business stakeholder for railcar products. A person in the role of a business stakeholder often is no user (i.e. actor) of the system.

Attributes

- No additional attributes.

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)
- weight (from Stakeholder)

Concrete Syntax and Mapping The notation for BusinessStakeholder is an entity notation. It uses the name attribute of the BusinessStakeholder metaclass to label its instances. The other attributes of BusinessStakeholder are currently not reflected in the notation.

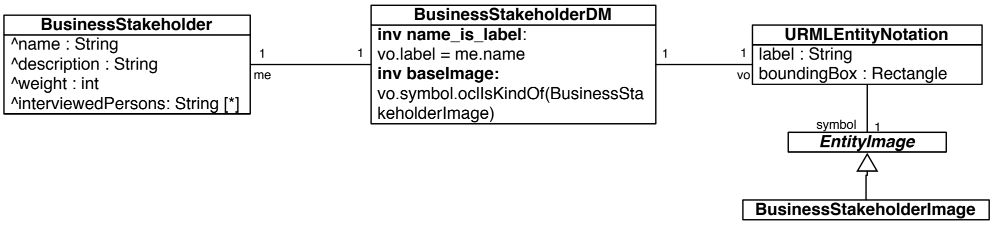


Figure A.18.: BusinessStakeholder Notation

Images BusinessStakeholderImage depicts two persons that are shaking hands. Both persons have a suitcase standing beneath them.

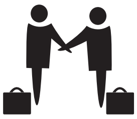


Figure A.19.: BusinessStakeholderImage

A.4.11. Customer (Class)

Somebody who is primarily interested in the system because he pays for it.

Generalizations

- Stakeholder

Description A special kind of stakeholder that purchases or funds the development of a system or process (e.g. hospital CFO). The term customer may also be used to describe an entire entity, e.g. an Integrated Healthcare Network. In such a scenario, the customer usually is no user of the system. In the private sector, a customer might also be the end user (e.g. the buyer of a text editing software).

Attributes

- No additional attributes.

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)
- weight (from Stakeholder)

Concrete Syntax and Mapping The notation for Customer is an entity notation. It uses the name attribute of the Customer meta-class to label its instances. The other attributes of Customer are currently not reflected in the notation.

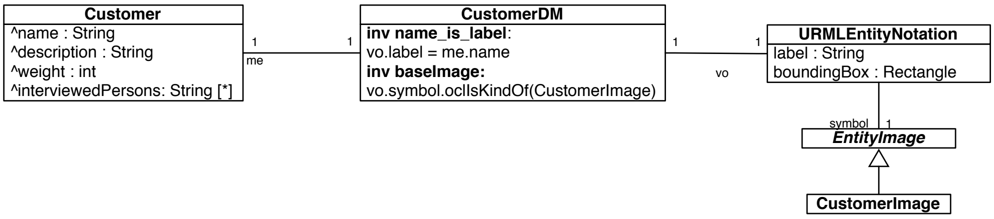


Figure A.20.: Customer Notation

Images CustomerImage depicts a person pushing a shopping cart.

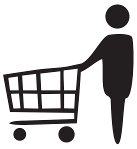


Figure A.21.: CustomerImage

A.4.12. Danger (Abstract Class)

The potential occurrence of something to be avoided, because it harms assets, persons, or systems.

Generalizations

- URMLModelEntity

Description Danger models the potential occurrence of something to be avoided as it may cause physical (see Hazard) or financial (see Threat) harm to the system, its stakeholders, or the assets of these stakeholders. A danger has a probability with which it occurs and a severity describing the magnitude of its impact. Dangers may threaten stakeholders, service providers, or assets, which we summarize under the term HarmedElement. In addition to harmed elements, also processes can be vulnerable to a danger, showing weak points of the system. While being vulnerable to danger, processes can also be the source of danger. Dangers should be mitigated by either appropriate procedures or requirements that the system under discussion should satisfy.

Attributes

- severity: DangerSeverity

- probability: DangerProbability

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)

A.4.13. DangerProbability (Enumeration)

Defines the probability of occurrence of a danger.

Enumeration Literals

- Certain
- Likely
- Possible
- Unlikely
- Rare

Description The enumeration literals for DangerProbability are taken from the CORAS meta-model (Braber et al. 2007). The definition of exact semantics of the literals in URML is left to the modeler or should be agreed upon with the stakeholders.

A.4.14. DangerSeverity (Enumeration)

Defines the severity of occurrence of a danger.

Enumeration Literals

- Catastrophic
- Major
- Moderate
- Minor
- Insignificant

Description The enumeration literals for DangerSeverity are taken from the CORAS meta-model (Braber et al. 2007). The definition of exact semantics of the literals in URML is left to the modeler or should be agreed upon with the stakeholders.

A.4.15. DangerTrigger (Relationship)

Relates a process with the dangers it might trigger.

Generalizations

- URMLModelRelationship

Description DangerTrigger models, from the viewpoint of a single process, which dangers a single trigger might cause. This means that dangers that might happen in conjunction are connected by one such relationship. Dangers that might happen at different points of the processes are connected via separate instances of this relationship.

Attributes

- No additional attributes

Role Names and Cardinalities

triggeredDangers: Danger[1..*]

triggeringProcess: Process[1]

Template Sentence "Process triggers Danger(s)"

Concrete Syntax and Mapping The notation for DangerTrigger is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at Asset, and the static label text 'triggers'.

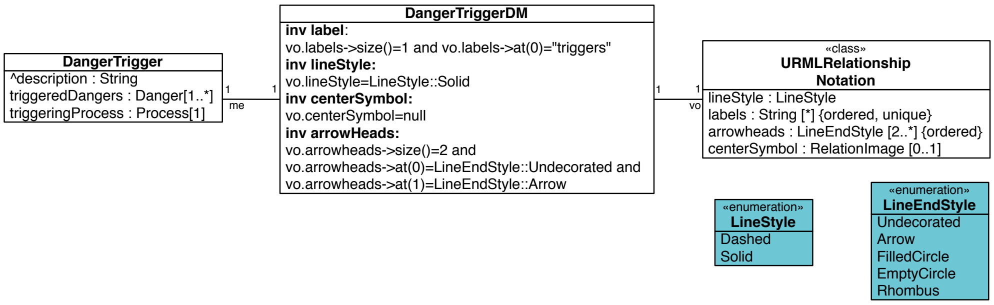

DangerTrigger
^description : String
triggeredDangers : Danger[1..*]
triggeringProcess : Process[1]
1
me
1
DangerTriggerDM
inv label:
vo.labels->size()=1 and vo.labels->at(0)="triggers"
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle::Undecorated and
vo.arrowheads->at(1)=LineEndStyle::Arrow
1
1
vo
«class»
URMLRelationship Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.22.: DangerTrigger Notation

Images No special imagery.

A.4.16. EntityObject (Class)

An object manipulated by the system.

Generalizations

- URMLModelEntity

Description An entity object can be something an actor gives to the system (e.g. name), something the system gives to the actor (e.g. an address) or something passed internally from one part of the system to another. An entity object can be a tangible object (e.g. a letter) or information. Entity objects are (as any other URML entity) decomposable. For an entity object instance, it is of particular interest to the modeler whether it is further decomposable or not. Entity objects are important for the system designers that receive a URML model when it is finished, as the processing of these objects strongly influences the architecture of the system. Thus, an entity object can be marked being atomic, i.e. non-decomposable. This is reflected then by a different notation.

Attributes

- atomic: Boolean

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)

Concrete Syntax and Mapping The notation for EntityObject is an entity notation. It uses the name attribute of the EntityObject meta-class to label its instances. The description attributeis currently not reflected in the notation. The atomic attribute switches between the image for atomic EntityObjects and the one for composite EntityObjects.

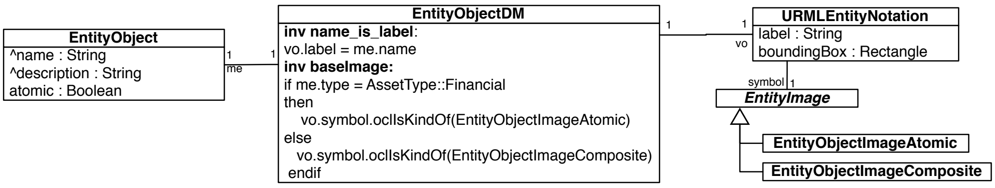


Figure A.23.: EntityObject Notation

Images EntityObjectImageAtomic depicts an open envelope. EntityObjectImageComposite depicts a filing cabinet.

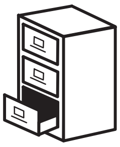


Figure A.24.: EntityObjectImage Composite

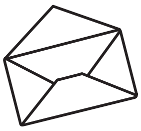


Figure A.25.: EntityObjectImage Atomic

A.4.17. EnvironmentProcess (Class)

A process being executed in the environment of the system under discussion.

Generalizations

- Process

Description As the system under discussion will be embedded into its environment, the activities of the environment need to be studied, in order to discover what the features of the system should be, and what interfaces it needs to offer. For danger modeling, it should be considered which environment processes should not interfere with, or which environment process could pose danger to the system. For business process modeling, environment processes that describe the customer environment are of particular interest. They are a description of activities as found in the customer's organization (often termed business processes). By the distinction between environment processes and use cases, the model shows how the system under discussion is used within its environment. When starting modeling, the environment process model can provide a black box view of the system. The business process aspect can be emphasized through the boolean attribute businessProcess inherited from the Process superclass.

Attributes

- No additional attributes

Inherited Attributes

- atomic (from Process)
- businessProcess (from Process)
- description (from URMLModelElement)
- name (from URMLModelEntity)
- preCondition (from Process)
- postCondition (from Process)
- underConstruction (from Process)

Concrete Syntax and Mapping The notation for EnvironmentProcess is an entity notation. It uses the name attribute of the EnvironmentProcess metaclass to label its instances. The description attribute is currently not reflected in the notation. Depending on the leaf and underConstruction attributes, overlay images are applied to to base image. If the leaf attribute is true, exactly one instance of LeafOverlayImage is applied to the base EnvironmentProcessImage, if false there must be no decoration of that kind. If the underConstruction attribute is true, exactly one instance of UnderConstructionOverlayImage is applied to the base EnvironmentProcessImage, if false there must be no decoration of that kind.

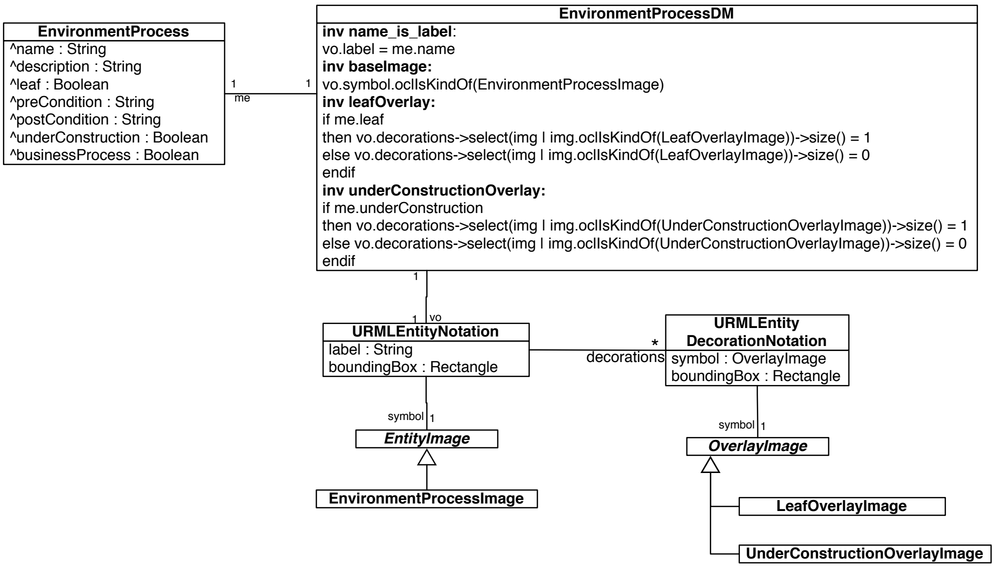

EnvironmentProcess
^name : String
^description : String
^leaf : Boolean
^preCondition : String
^postCondition : String
^underConstruction : Boolean
^businessProcess : Boolean
1
me
1
EnvironmentProcessDM
inv name_is_label:
vo.label = me.name
inv baseImage:
vo.symbol.oclIsKindOf(EnvironmentProcessImage)
inv leafOverlay:
if me.leaf
then vo.decorations->select(img | img.oclIsKindOf(LeafOverlayImage))->size() = 1
else vo.decorations->select(img | img.oclIsKindOf(LeafOverlayImage))->size() = 0
endif
inv underConstructionOverlay:
if me.underConstruction
then vo.decorations->select(img | img.oclIsKindOf(UnderConstructionOverlayImage))->size() = 1
else vo.decorations->select(img | img.oclIsKindOf(UnderConstructionOverlayImage))->size() = 0
endif
1
1
vo
URMLEntityNotation
label : String
boundingBox : Rectangle
*
decorations
URMLEntity
DecorationNotation
symbol : OverlayImage
boundingBox : Rectangle
symbol
1
EntityImage
EnvironmentProcessImage
symbol
1
OverlayImage
LeafOverlayImage
UnderConstructionOverlayImage

Figure A.26.: EnvironmentProcess Notation

Images The final image depicting EnvironmentProcess consists of a base image that is optionally decorated by overlays. The base image, EnvironmentProcessImage, depicts an oval inspired by the UML use case notation, in which two persons are depicted that shake hands. In between them stands a suitcase. Below them are two stylized arrows pointing in opposite directions. If the leaf attribute is true, LeafOverlayImage (depicting a filled black circle) is placed on the lower right of the base image. If the underConstruction attribute is true, UnderConstructionOverlayImage (depicting a yellow-and-black striped road barrier) is placed on the lower center.

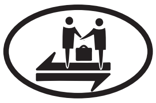


Figure A.27.: EnvironmentProcessImage

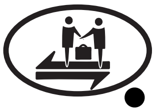


Figure A.28.: EnvironmentProcessImage with LeafOverlayImage

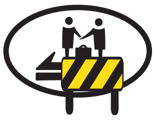


Figure A.29.: EnvironmentProcessImage with UnderConstructionOverlayImage

A.4.18. ExternalLink (Class)

Provides meta-data for traceability links.

Generalizations None

Description In tools that implement this class, ExternalLink instances can be added to any URMLModelElement to provide traceability meta-data that point to other models or tools. Such links are not part of URML's notation, they should be visualized in a separate section, wherever the tool visualizes the attributes of a model element. The short name provides a human-readable string, and the URI specifies the link target.

Attributes

- URI : String
- shortName : String
- modelElement : URMLModelElement

Inherited Attributes None

Concrete Syntax and Mapping None

Images None

A.4.19. Feature (Class)

Stakeholder-visible characteristic of the system under discussion.

Generalizations

- AbstractFeature

Description A feature is a desirable or existing, prominent or distinctive stakeholder-visible aspect, quality, or characteristic of a system. A feature enables environment processes, i.e. it is a necessary component for their execution. A feature contributes to a goal if the user visible property helps the stakeholder reach that goal. A set of features defines a product. A feature makes a set of functional requirements visible to stakeholders. Quality requirements describe constraints that the feature must satisfy. A feature that is in every product of a product line can be marked being a core feature.

Attributes

- core: Boolean

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)

Concrete Syntax and Mapping The notation for Feature is an entity notation. It uses the name attribute of the Feature meta-class to label its instances. The description attribute is currently not reflected in the notation. Depending on the core attribute, an overlay image is applied to to base image. If the core attribute is true, exactly one instance of CoreOverlayImage is applied to the base FeatureImage, if false there must be no decoration of that kind.

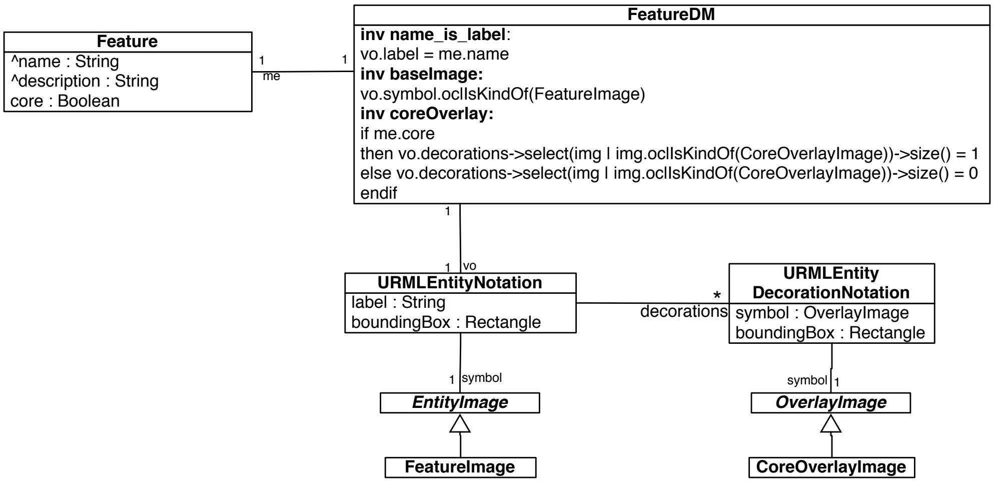


Figure A.30.: Feature Notation

Images The complete symbol for Feature consists of a base image that is possibly decorated by one overlay if the value of the core attribute is true. The base, FeatureImage depicts a puzzle piece. CoreOverlayImage depicts an apple's core. If applicable, it is placed on the right of the puzzle piece.

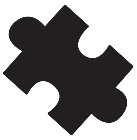


Figure A.31.: FeatureImage

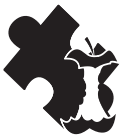


Figure A.32.: FeatureImage with CoreOverlayImage

A.4.20. FeatureConstraint (Relationship)

Maps a quality requirement to a feature.

Generalizations

- URMLModelRelationship

Description A feature that itself is not a user-visible quality of the system might still be constrained by a quality requirement. That is, the quality requirement is not advertised as a feature itself, but as a constraint to a feature (e.g. Feature is “Long Battery Runtime”, QualityRequirement is “˜8hours”).

Attributes

- No additional attributes

Role Names and Cardinalities

constraint: QualityRequirement[1]

constrainedFeature: Feature[1]

Template Sentence “QualityRequirement constrains Feature”

Concrete Syntax and Mapping The notation for FeatureConstraint is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at Feature, and the static label text ’constrains’.

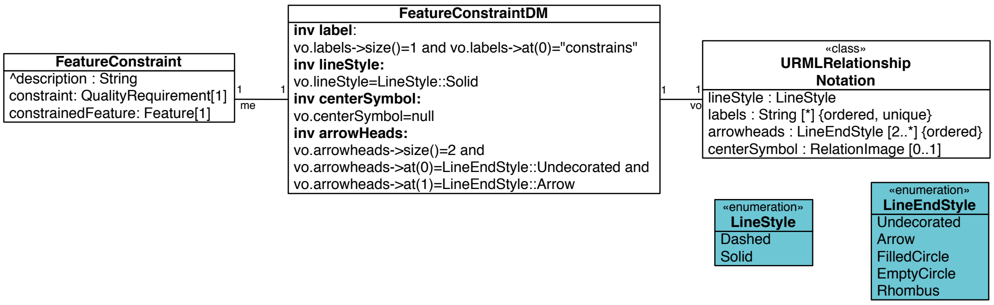

FeatureConstraint
^description : String
constraint: QualityRequirement[1]
constrainedFeature: Feature[1]
1
me
1
FeatureConstraintDM
inv label:
vo.labels->size()=1 and vo.labels->at(0)="constrains"
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle::Undecorated and
vo.arrowheads->at(1)=LineEndStyle::Arrow
1
1
vo
«class»
URMLRelationship
Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.33.: FeatureConstraint Notation

Images No special imagery.

A.4.21. FeatureDescriptionRequirement (Relationship)

Maps a functional requirement to a feature.

Generalizations

- URMLModelRelationship

Description A feature just describes what is a distinctive characteristic of the system (e.g. what the user sees from the outside). This has to be backed by functional requirements. Which feature is described by which functional requirements is shown through instances of this relationship.

Attributes

- No additional attributes

Role Names and Cardinalities

detailingRequirement: FunctionalRequirement[1]

detailedFeature: Feature[1]

Template Sentence “FunctionalRequirement details Feature”

Concrete Syntax and Mapping The notation for FeatureDescriptionRequirement is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at Feature, and the static label text ’details’.

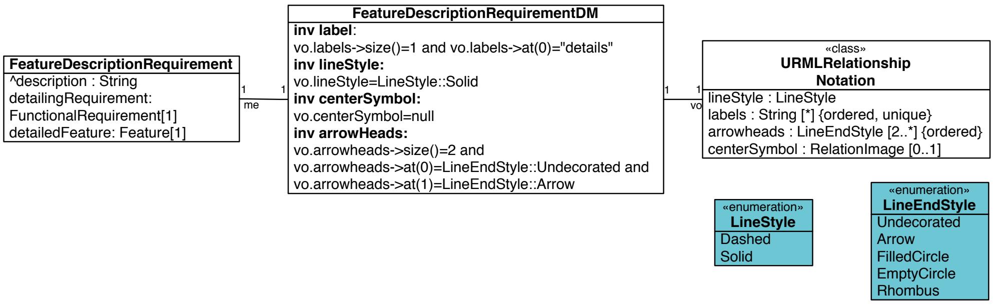

FeatureDescriptionRequirement
^description : String
detailingRequirement:
FunctionalRequirement[1]
detailedFeature: Feature[1]
1
1
me
FeatureDescriptionRequirementDM
inv label:
vo.labels->size()=1 and vo.labels->at(0)="details"
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle::Undecorated and
vo.arrowheads->at(1)=LineEndStyle::Arrow
1
1
vo
«class»
URMLRelationship
Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.34.: FeatureDescriptionRequirement Notation

Images No special imagery.

A.4.22. FeatureDescriptionUseCase (Relationship)

Maps a use case to a feature.

Generalizations

- URMLModelRelationship

Description This relationship enables a mapping of features to system behavior (i.e. use cases). While the FeatureDescriptionRequirement relationship allows to map functional requirements to features, the modeler still does not now which functions of the system might be invoked in which order.

Attributes

- No additional attributes

Role Names and Cardinalities

detailingUseCase: UseCase[1]

detailedFeature: Feature[1]

Template Sentence “UseCase details Feature”

Concrete Syntax and Mapping The notation for FeatureDescriptionUseCase is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at Feature, and the static label text ’details’.

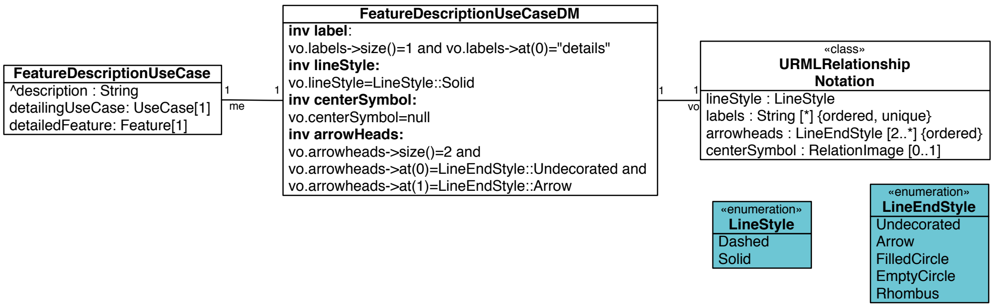

FeatureDescriptionUseCase
^description : String
detailingUseCase: UseCase[1]
detailedFeature: Feature[1]
1
me
1
FeatureDescriptionUseCaseDM
inv label:
vo.labels->size()=1 and vo.labels->at(0)="details"
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle::Undecorated and
vo.arrowheads->at(1)=LineEndStyle::Arrow
1
1
vo
«class»
URMLRelationship Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.35.: FeatureDescriptionUseCase Notation

Images No special imagery.

A.4.23. FeatureExclusion (Relationship)

Expresses that the selection of one feature leads to the exclusion of another feature.

Generalizations

- URMLModelRelationship

Description Some features or feature groups may not be part of the same product, while they are still part of the same product line. This relationship expresses that the selection of one feature or feature group excludes the selection of the other feature or feature group.

Attributes

- No additional attributes

Role Names and Cardinalities

excludingFeature: AbstractFeature[1]

excludedFeature: AbstractFeature[1]

Template Sentence "AbstractFeature excludes AbstractFeature"

Concrete Syntax and Mapping The notation for FeatureExclusion is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at excludedFeature, and the static label text 'excludes'.

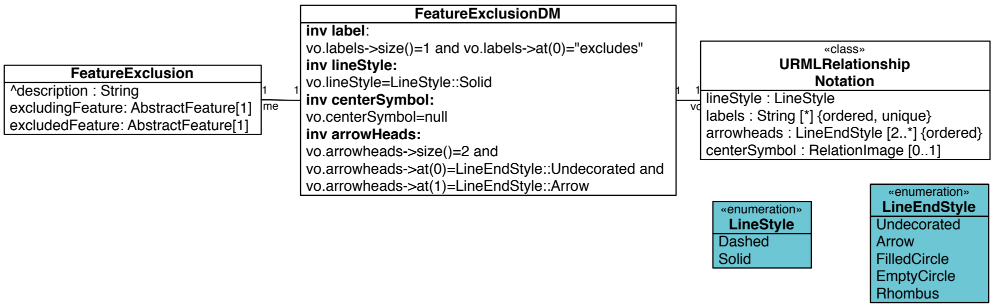

FeatureExclusion
^description : String
excludingFeature: AbstractFeature[1]
excludedFeature: AbstractFeature[1]
1
me
1
FeatureExclusionDM
inv label:
vo.labels->size()=1 and vo.labels->at(0)="excludes"
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle::Undecorated and
vo.arrowheads->at(1)=LineEndStyle::Arrow
1
1
vo
«class»
URMLRelationship
Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.36.: FeatureExclusion Notation

Images No special imagery.

A.4.24. FeatureGroup (Class)

A set of related features from which a product designer would select a subset according to a defined selection rule.

Generalizations

- AbstractFeature

Description A feature group is a set of related features, e.g. “power options”. A feature group is nothing 'physical', it is used for grouping purposes. For the product line modeler it is nevertheless very important as it is the central concept for modeling a feature tree. The topmost group of the feature tree is marked with root == true. The tree structure is enabled through the composite pattern with AbstractFeature and Feature. A product line can have exactly one feature tree and thus points exactly to one FeatureGroup with root == true.

Attributes

- root: Boolean
- underConstruction: Boolean
- selectionType: FeatureSelectionType

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)

Concrete Syntax and Mapping The notation for Feature is an entity notation. It uses the name attribute of the Feature meta-class to label its instances. The description attribute is currently not reflected in the notation. Which base image is used is determined by the root attribute: if root is true, FeatureTreeImage is used, FeatureGroupImage otherwise. Depending on the underConstruction attribute, an overlay image is applied to to base image. If the it is true, exactly one instance of UnderConstructionOverlayImage is applied to the base image, if false there must be no decoration of that kind. Depending on the selectionType attribute, an additional overlay may be added.

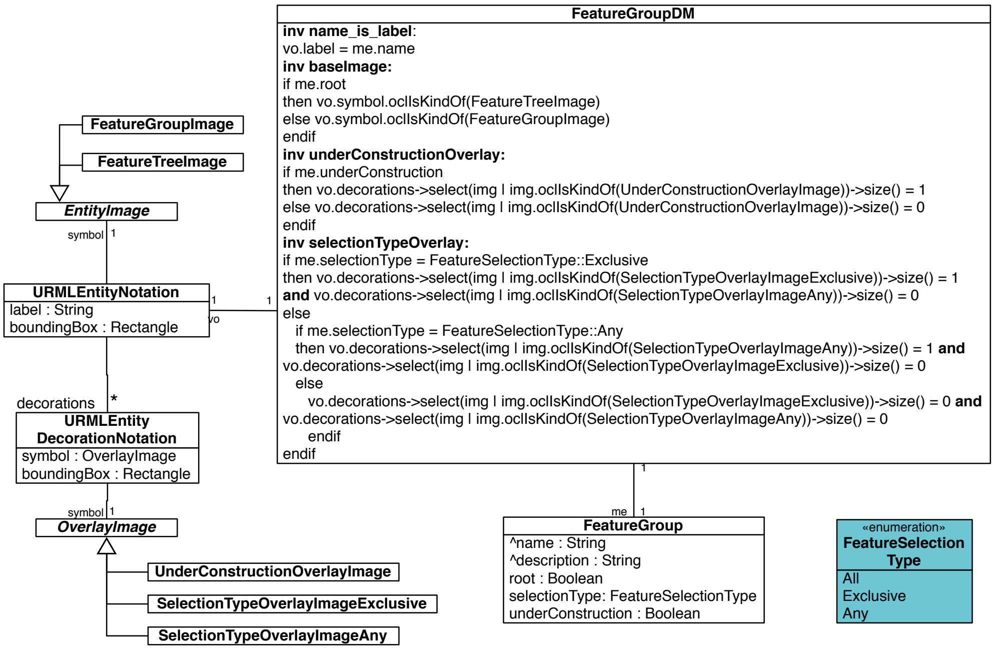

FeatureGroupDM
inv name_is_label:
vo.label = me.name
inv baseImage:
if me.root
then vo.symbol.oclIsKindOf(FeatureTreeImage)
else vo.symbol.oclIsKindOf(FeatureGroupImage)
endif
inv underConstructionOverlay:
if me.underConstruction
then vo.decorations->select(img I img.oclIsKindOf(UnderConstructionOverlayImage))->size() = 1
else vo.decorations->select(img I img.oclIsKindOf(UnderConstructionOverlayImage))->size() = 0
endif
inv selectionTypeOverlay:
if me.selectionType = FeatureSelectionType::Exclusive
then vo.decorations->select(img I img.oclIsKindOf(SelectionTypeOverlayImageExclusive))->size() = 1
and vo.decorations->select(img I img.oclIsKindOf(SelectionTypeOverlayImageAny))->size() = 0
else
if me.selectionType = FeatureSelectionType::Any
then vo.decorations->select(img I img.oclIsKindOf(SelectionTypeOverlayImageAny))->size() = 1 and
vo.decorations->select(img I img.oclIsKindOf(SelectionTypeOverlayImageExclusive))->size() = 0
else
vo.decorations->select(img I img.oclIsKindOf(SelectionTypeOverlayImageExclusive))->size() = 0 and
vo.decorations->select(img I img.oclIsKindOf(SelectionTypeOverlayImageAny))->size() = 0
endif
endif
FeatureGroupImage
FeatureTreeImage
EntityImage
symbol
1
URMLEntityNotation
label : String
boundingBox : Rectangle
1
vo
1
decorations
*
URMLEntity DecorationNotation
symbol : OverlayImage
boundingBox : Rectangle
symbol
1
OverlayImage
UnderConstructionOverlayImage
SelectionTypeOverlayImageExclusive
SelectionTypeOverlayImageAny
1
me
1
FeatureGroup
^name : String
^description : String
root : Boolean
selectionType: FeatureSelectionType
underConstruction : Boolean
«enumeration»
FeatureSelection Type
All
Exclusive
Any

Figure A.37.: FeatureGroup Notation

Images The overall image representing a FeatureGroup instance consists of a base image that is optionally decorated by two overlay images depending on the attributes underConstruction and selectionType. The base image depends on the root attribute: if it is true, the base image is FeatureTreeImage. It depicts a tree whose leaves are puzzle pieces. The puzzle pieces stand for the features (see section A.4.19). In the other case, the base image is FeatureGroupImage. It depicts a set of six puzzle pieces, of which two and four are connected. UnderConstructionOverlayImage depicts a yellow-and-black striped road barrier. It is applied if the underConstruction attribute is set to true.
Depending on the value of the selectionType attribute, an additional overlay depicting an arc added. If the value is All, no arc is drawn. If the value is Any, a hollow arc is drawn. If the value is 'Exclusive', the arc is filled with black color. The arc connects (and possibly hides part of) the lines constituting the notation of the incoming feature selection option relationships.

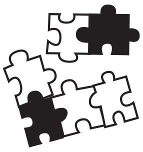


Figure A.38.: FeatureGroupImage

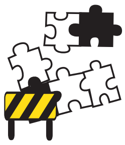


Figure A.39.: FeatureGroupImage with Under-Construction OverlayImage

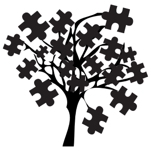


Figure A.40.: FeatureTreeImage

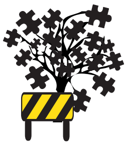


Figure A.41.: FeatureTreeImage with Under-Construction OverlayImage

A.4.25. FeatureList (Relationship)

Shows to which product a feature belongs.

Generalizations

- URMLModelRelationship

Description This relationship enables modeling which of the potential features of a product line are actually mapped to a product. Each feature that was selected for the product is connected to it via this relationship.

Attributes

- No additional attributes

Role Names and Cardinalities

product:	Product[1]
component:	Feature[1]

Template Sentence “Product has Feature”

Concrete Syntax and Mapping The notation for FeatureList is a relationship notation. It has a solid line, no center symbol, an rhombus-style arrowhead at the end of line ending at Product, and no static label text.

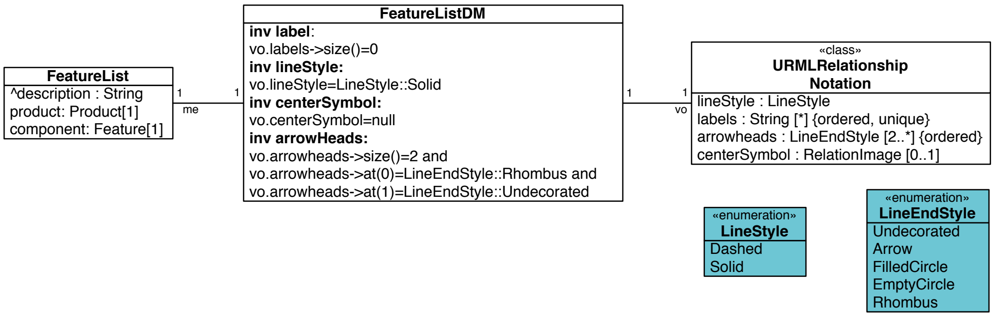


Figure A.42.: FeatureList Notation

Images No special imagery.

A.4.26. FeatureRequirement (Relationship)

Shows that a feature requires another feature.

Generalizations

- URMLModelRelationship

Description Some features or feature groups may have other features as a prerequisite. This can in turn be another feature or feature group. The requiring feature may only be selected from a product line if the required feature is chosen as well.

Attributes

- No additional attributes

Role Names and Cardinalities

requiringFeature: AbstractFeature[1]

requiredFeature: AbstractFeature[1]

Template Sentence “AbstractFeature requires AbstractFeature”

Concrete Syntax and Mapping The notation for FeatureRequirement is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at requiredFeature, and the static label text ’presupposes’.

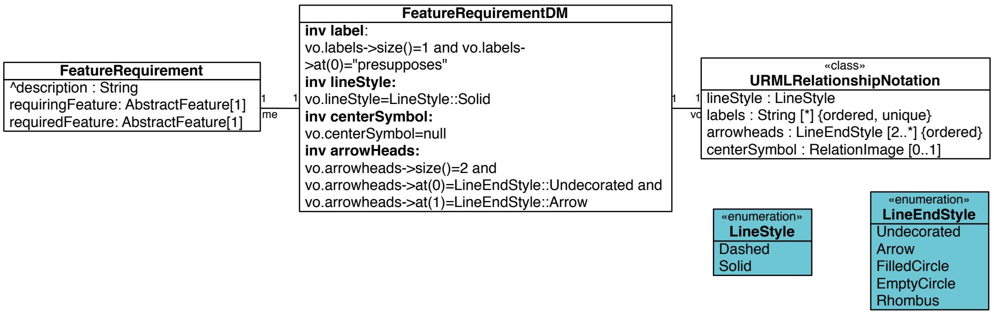

FeatureRequirement
^description : String
requiringFeature: AbstractFeature[1]
requiredFeature: AbstractFeature[1]
1
me
1
FeatureRequirementDM
inv label:
vo.labels->size()=1 and vo.labels->at(0)="presupposes"
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle::Undecorated and
vo.arrowheads->at(1)=LineEndStyle::Arrow
1
1
vo
«class»
URMLRelationshipNotation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.43.: FeatureRequirement Notation

Images No special imagery.

A.4.27. FeatureSelectionOption (Relationship)

Shows to which feature group a feature or feature group belongs.

Generalizations

- URMLModelRelationship

Description A feature group prescribes a rule with which sub-features or sub-feature groups of the group may be selected. These selectable features or feature groups are connected to the parent feature group via this relationship. The relationship may further confine the selection rule by specifying that the connected feature or feature group is mandatory and must be selected. This confinement only makes sense for feature groups of selection types “Any” or “Exclusive”. The “All” selection type already prescribes that all sub-features must be selected. For the “Any” selection type the confinement means that the mandatory feature or feature group must be selected plus any combinations of the sibling features or feature groups. For the “Exclusive” selection type, the confinement only makes sense if there is more than one sibling in addition to the mandatory feature or feature group. The confinement means then that the mandatory feature or feature group must be selected plus exactly one of the other features or feature groups.

Attributes

- mandatory: Boolean

Role Names and Cardinalities

selectionGroup:	FeatureGroup[1]
option:	Feature[1]

Template Sentence “FeatureGroup has option Feature”

Concrete Syntax and Mapping The notation for FeatureSelectionOption is a relationship notation. It has a solid line, no center symbol, and no static label text. The arrow-style arrowhead at the end of line ending at Feature is determined by the value of the mandatory attribute. If the value is true, FilledCircle results as an arrowhead, and EmptyCircle results if the value is false.

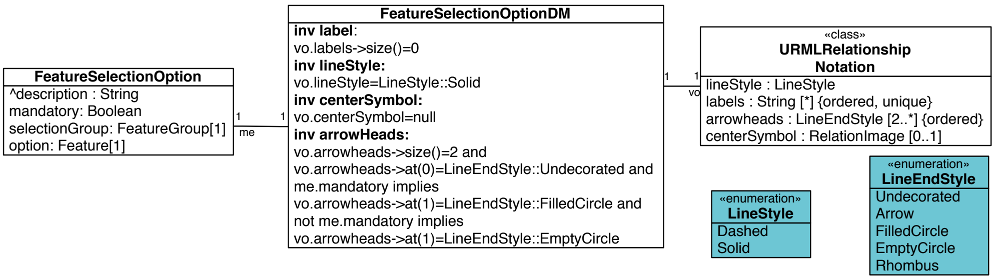


Figure A.44.: FeatureSelectionOption Notation

Images No special imagery.

A.4.28. FeatureSelectionType (Enumeration)

Categorized Class

- FeatureSelectionOptions

Enumeration Literals

- All
- Any
- Exclusive

Description

All - All sub-features or feature groups must be selected.

Any - Any combination of features or feature groups can be selected.

Exclusive - Exactly one of the features or feature groups can be selected.

A.4.29. FeatureTree (Relationship)

Links a product line with its tree of features, rooted in a feature group.

Generalizations

- URMLModelRelationship

Description For a given tree of features, there should be one feature group at the top of the tree that is marked being the root. This root feature group can then be linked to the product line it shall represent. This is an exclusive relationship; a product line may link to exactly one feature group and vice versa.

Attributes

- No additional attributes

Role Names and Cardinalities

representedProductLine: ProductLine[1]

featureTreeRoot: FeatureGroup[1]

Template Sentence “ProductGroup represents ProductLine”

Constraints `only_root_group: featureGroup.root == true`

A feature tree relationship may only relate to a feature group that is marked being a root.

Concrete Syntax and Mapping The notation for FeatureTree is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at FeatureGroup, and the static label text 'represents'.

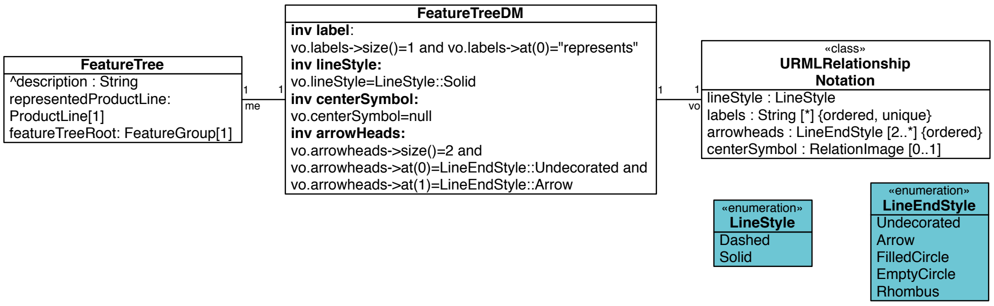

FeatureTree
^description : String
representedProductLine:
ProductLine[1]
featureTreeRoot: FeatureGroup[1]
1
me
1
FeatureTreeDM
inv label:
vo.labels->size()=1 and vo.labels->at(0)="represents"
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle::Undecorated and
vo.arrowheads->at(1)=LineEndStyle::Arrow
1
1
vo
«class»
URMLRelationship
Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.45.: FeatureTree Notation

Images No special imagery.

A.4.30. FunctionalRequirement (Class)

A requirement that demands a function of the system.

Generalizations

- Requirement

Description A functional requirement is a kind of requirement describing required functionality of the system under discussion, as opposed to a quality requirement. Processes require certain functions of the system, whose requirement is modeled through functional requirements. Functional requirements detail features such that they manifest the set of functional requirements that are made visible by that feature. Functional requirements are not necessarily stakeholder-visible.

Attributes

- No additional attributes

Inherited Attributes

- costDriver (from Requirement)
- description (from URMLModelElement)
- name (from URMLModelElement)
- priority (from Requirement)
- rank (from Requirement)
- regulatory (from Requirement)

Concrete Syntax and Mapping The notation for FunctionalRequirement is an entity notation. It uses the name attribute of the FunctionalRequirement meta-class to label its instances. Depending on the regulatory attribute, an overlay image is applied to to base image. If the regulatory attribute is true, exactly one instance of RegulatoryOverlayImage is applied to the base FunctionalRequirementImage, if false there must be no decoration of that kind. The other attributes of FunctionalRequirement are currently not reflected in the notation.

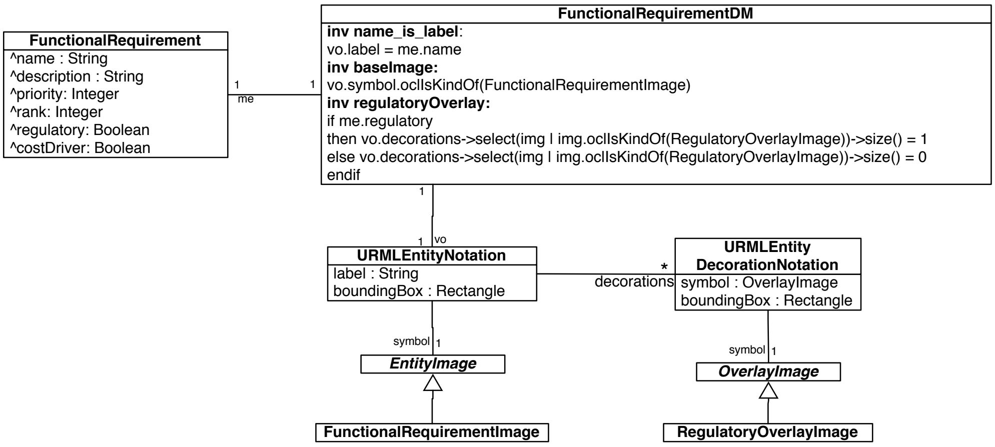


Figure A.46.: FunctionalRequirement Notation

Images The final symbol representing instances of FunctionalRequirement consists of a base image that is optionally decorated with an overlay image. The base image, FunctionalRequirementImage, depicts two gears that mesh. The overlay, RegulatoryOverlayImage, depicts a traffic policeman.

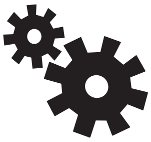


Figure A.47.: FunctionalRequirementImage

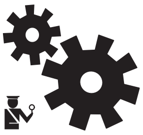


Figure A.48.: FunctionalRequirementImage with RegulatoryOverlayImage

A.4.31. Goal (Abstract Class)

Abstract base class for taking part in relationships that are common to hard goals and soft goals.

Generalizations

- URMLModelEntity

Description A goal is expressed by one to many stakeholders, expressing “a condition or state of affairs in the world that [they] would like to achieve” (ITU 2008a). A goal is product independent. Goals do not exist in isolation; therefore they can be connected via two relationships, namely GoalContribution and GoalDecomposition. Goals are either hard or soft. The distinction between HardGoal and SoftGoal depends on whether there is a description of how to measure success in reaching the goal (see AssessmentSketch). Goals can be reason for stakeholder requests. In the system under discussion goals are realized by features.

Attributes

- No additional attributes.

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)

A.4.32. GoalContribution (Relationship)

The positive or negative impact the achievement of one goal has on the achievement of another goal.

Generalizations

- URMLModelRelationship

Description The GoalContribution is a relationship between goals that is used to express how goals positively or negatively affect each other. The goal contribution relationship has an attribute signifying the strength of the contribution.

Attributes

- type: GoalContributionType

Role Names and Cardinalities

affectedGoal: Goal[1]

contributingGoal: Goal[1]

Template Sentence "Goal contributes to Goal"

Constraints The contribution might be known while its value is unknown. An unknown contribution should however only be allowed in early phases of the requirement engineering process as it leads to ambiguity as long as it is unknown.

Concrete Syntax and Mapping The notation for GoalContribution is a relationship notation. It has a dashed line, no center symbol, an arrow-style arrowhead at the end of line ending at affectedGoal, and a dynamic label text depending on the value of the GoalContribution's type attribute.

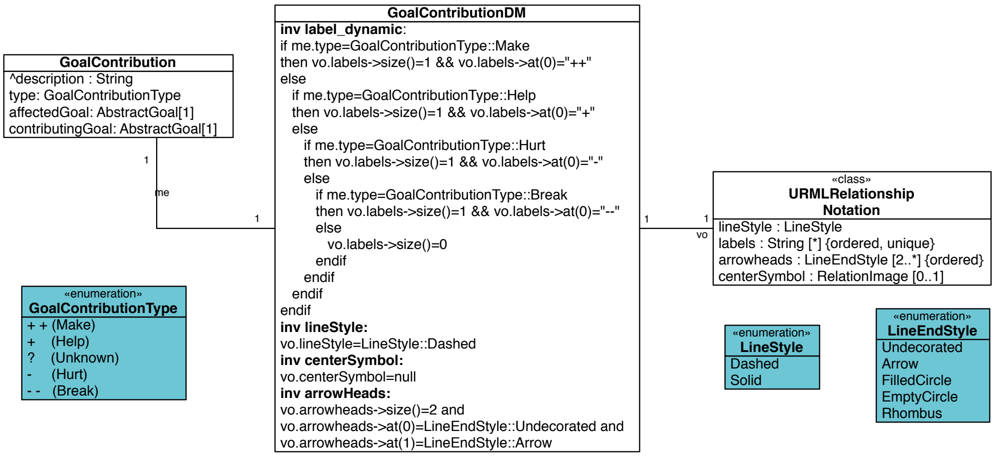

GoalContribution
^description : String
type: GoalContributionType
affectedGoal: AbstractGoal[1]
contributingGoal: AbstractGoal[1]
1
me
1
GoalContributionDM
inv label_dynamic:
if me.type=GoalContributionType::Make
then vo.labels->size()=1 && vo.labels->at(0)="++"
else
if me.type=GoalContributionType::Help
then vo.labels->size()=1 && vo.labels->at(0)="+"
else
if me.type=GoalContributionType::Hurt
then vo.labels->size()=1 && vo.labels->at(0)="-"
else
if me.type=GoalContributionType::Break
then vo.labels->size()=1 && vo.labels->at(0)="--"
else
vo.labels->size()=0
endif
endif
endif
endif
inv lineStyle:
vo.lineStyle=LineStyle::Dashed
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle::Undecorated and
vo.arrowheads->at(1)=LineEndStyle::Arrow
1
1
vo
«class»
URMLRelationship Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
GoalContributionType
+ + (Make)
+ (Help)
? (Unknown)
- (Hurt)
- - (Break)
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.49.: GoalContribution Notation

Images No special imagery.

A.4.33. GoalContributionType (Enumeration)

Categorized Class

- GoalContribution

Enumeration Literals

- ++ (make)
- + (help)
- ? (unknown)
- - (hurt)
- – (break)

Description GoalContributionType enumerates the different kinds of goal contribution. A contribution can be positive (make, help), unknown, or negative (hurt, break).

A.4.34. GoalDecomposition (Relationship)

Establishes a parent goal - subgoal relationship.

Generalizations

- URMLModelRelationship

Description A GoalDecomposition allows to model a hierarchy of goals. The hierarchy can be mixed, i.e. consist of Goal as well as SoftGoal instances.

Attributes

- No additional attributes.

Role Names and Cardinalities

compositeGoal: Goal[1]

componentGoal: Goal[1]

Template Sentence "Goal has subgoal Goal"

Concrete Syntax and Mapping The notation for GoalDecomposition is a relationship notation. It has a solid line, no center symbol, an rhombus-style arrowhead at the end of line ending at compositeGoal, and no label.

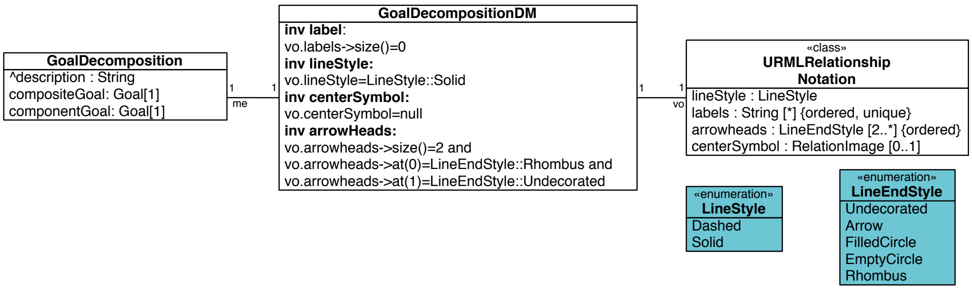


Figure A.50.: GoalDecomposition Notation

Images No special imagery.

A.4.35. GoalRealization (Relationship)

Maps a feature to the goal it realizes.

Generalizations

- URMLModelRelationship

Description In order to show stakeholders how the system will support their goals, features of the system are mapped to goals. This relationships shows which goal is realized by which feature.

Attributes

- No additional attributes.

Role Names and Cardinalities

addressedGoal: Goal[1]

realizingFeature: Feature[1]

Template Sentence “Feature realizes Goal”

Concrete Syntax and Mapping The notation for GoalRealization is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at Goal, and the static label text ’realizes’.

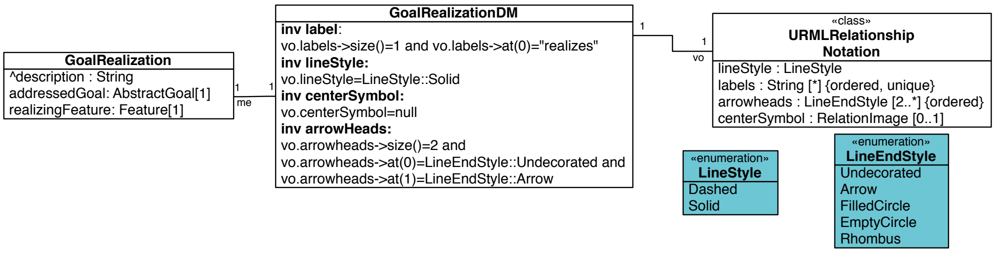

GoalRealization
^description : String
addressedGoal: AbstractGoal[1]
realizingFeature: Feature[1]
1
me
1
GoalRealizationDM
inv label:
vo.labels->size()=1 and vo.labels->at(0)="realizes"
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle::Undecorated and
vo.arrowheads->at(1)=LineEndStyle::Arrow
1
1
vo
«class»
URMLRelationship Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.51.: GoalRealization Notation

Images No special imagery.

A.4.36. GoalStatement (Relationship)

Shows which stakeholder expressed a goal.

Generalizations

- URMLModelRelationship

Description This relationship enables modeling which stakeholder expressed a goal.

Attributes

- No additional attributes.

Role Names and Cardinalities

issuingStakeholder: Stakeholder[1]

statedGoal: Goal[1]

Template Sentence “Stakeholder expresses Goal”

Concrete Syntax and Mapping The notation for GoalStatement is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at Goal, and the static label text ’realizes’.

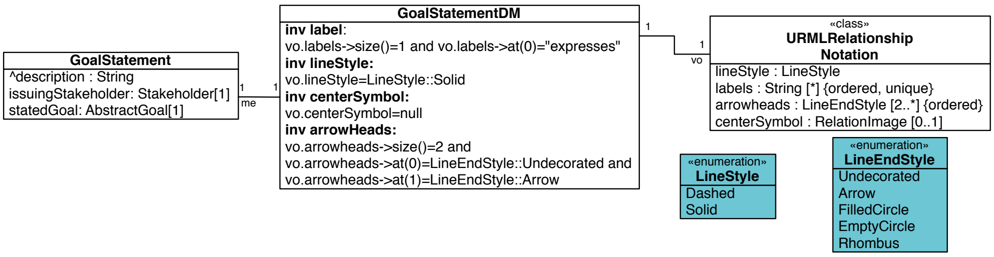

GoalStatement
^description : String
issuingStakeholder: Stakeholder[1]
statedGoal: AbstractGoal[1]
1
me
1
GoalStatementDM
inv label:
vo.labels->size()=1 and vo.labels->at(0)="expresses"
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle::Undecorated and
vo.arrowheads->at(1)=LineEndStyle::Arrow
1
1
vo
«class»
URMLRelationship
Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.52.: GoalStatement Notation

Images No special imagery.

A.4.37. HardGoal (Class)

A specialization of goal whose achievement by the system can be tested.

Generalizations

- Goal

Description An objective expressed by a stakeholder that should be achieved by the system in operation. A goal's realization is testable/measurable in some way (as opposed to SoftGoal).

Attributes

- No additional attributes

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)

Concrete Syntax and Mapping The notation for HardGoal is an entity notation. It uses the name attribute of the HardGoal meta-class to label its instances. The description attribute is currently not reflected in the notation.

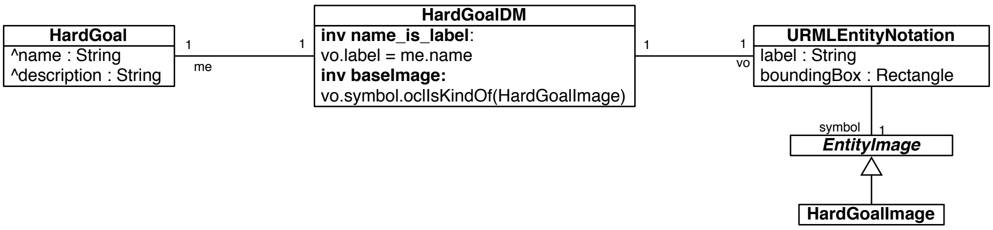


Figure A.53.: HardGoal Notation

Images The notation uses exactly one image, HardGoalImage. It depicts a simplified target with an arrow sticking in the bull's eye.

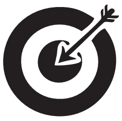


Figure A.54.: HardGoalImage

A.4.38. Harm (Relationship)

Shows which danger would harm an harmed element upon occurrence.

Generalizations

- URMLModelRelationship

Description This relationship helps modeling the impact of dangers would they occur. It connects dangers with harmed elements. As a rule of thumb, the more stakeholders are involved, the more catastrophic the impact of the danger.

Attributes

- No additional attributes.

Role Names and Cardinalities

danger:	Danger[1]
harmedElement:	HarmedElement[1]

Template Sentence “Danger endangers HarmedElement”

Concrete Syntax and Mapping The notation for Harm is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at HarmedElement, and the static label text ’harms’.

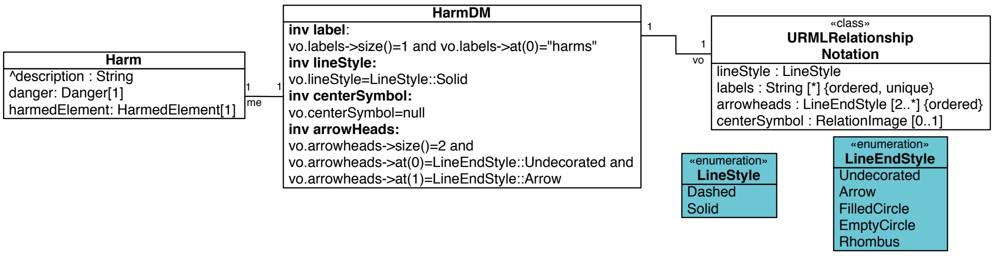

Harm
^description : String
danger: Danger[1]
harmedElement: HarmedElement[1]
1
me
1
HarmDM
inv label:
vo.labels->size()=1 and vo.labels->at(0)="harms"
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle::Undecorated and
vo.arrowheads->at(1)=LineEndStyle::Arrow
1
1
vo
«class»
URMLRelationship Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.55.: Harm Notation

Images No special imagery.

A.4.39. HarmedElement (Abstract Class)

A superclass of stakeholder, asset, and service provider to show these can be affected by danger.

Generalizations

- URMLModelEntity

Description Generalization of all entities that may be affected by danger (Stakeholder, Asset, Service Provider). Harmed elements can be protected by procedures or requirements. The relationships connecting Danger, HarmedElement, and the mitigating entity are called either procedural mitigation (in the case of a procedure) or requirement mitigation (in the case of a requirement).

Attributes

- No additional attributes.

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)

A.4.40. Hazard (Class)

Generalizations

- Danger

Description A type of Danger; Potential cause of physical harm or injury to stakeholders, service providers, or assets. Hazards are typed by their source (e.g. electrical, mechanical).

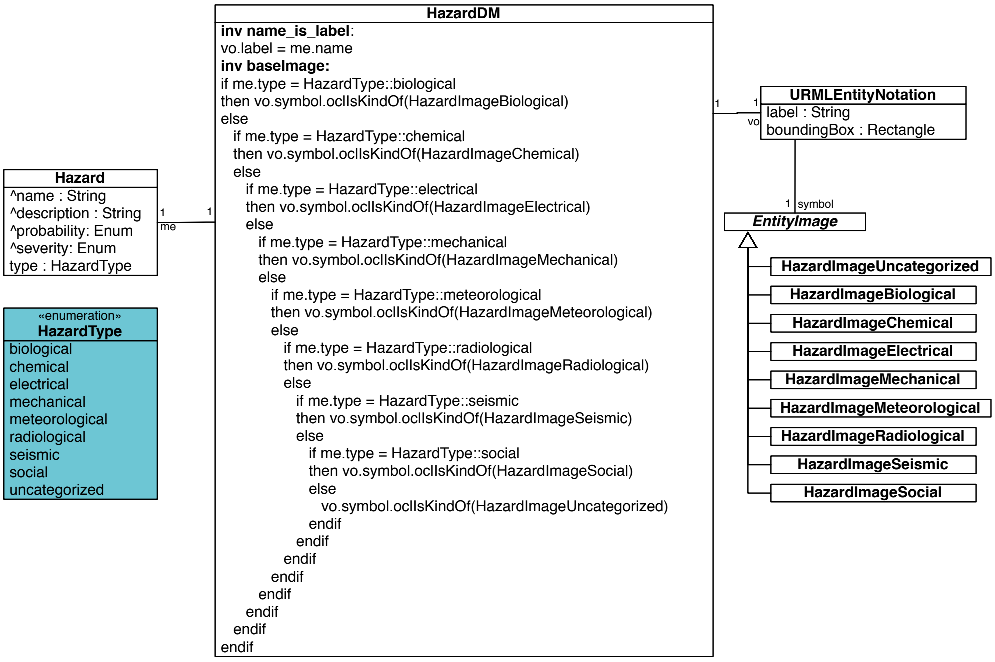
Attributes

- type: HazardType

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)
- probability (from Danger)
- severity (from Danger)

Concrete Syntax and Mapping The notation for Hazard is an entity notation. It uses the name attribute of the Hazard meta-class to label its instances. The image used to depict the Hazard instance depends on the value of the type attribute. The other attributes are currently not reflected in the notation.



```
HazardDM
inv name_is_label:
vo.label = me.name
inv baseImage:
if me.type = HazardType::biological
then vo.symbol.oclIsKindOf(HazardImageBiological)
else
  if me.type = HazardType::chemical
  then vo.symbol.oclIsKindOf(HazardImageChemical)
  else
    if me.type = HazardType::electrical
    then vo.symbol.oclIsKindOf(HazardImageElectrical)
    else
      if me.type = HazardType::mechanical
      then vo.symbol.oclIsKindOf(HazardImageMechanical)
      else
        if me.type = HazardType::meteorological
        then vo.symbol.oclIsKindOf(HazardImageMeteorological)
        else
          if me.type = HazardType::radiological
          then vo.symbol.oclIsKindOf(HazardImageRadiological)
          else
            if me.type = HazardType::seismic
            then vo.symbol.oclIsKindOf(HazardImageSeismic)
            else
              if me.type = HazardType::social
              then vo.symbol.oclIsKindOf(HazardImageSocial)
              else
                vo.symbol.oclIsKindOf(HazardImageUncategorized)
              endif
            endif
          endif
        endif
      endif
    endif
  endif
endif
```

Figure A.56.: Hazard Notation

Images The images used for representing Hazard instances share a common theme: Any image is base on the same triangle with rounded corners. That triangle is filled with different content depending on the value of the type attribute. If the value of type is uncategorized, an exclamation mark is within the borders of the triangle. If it is 'Biological', 'Chemical', 'Electrical', 'Mechanical', or 'Radiological', the content is adapted from the international standard on safety colors and safety signs, ISO 7010 (ISO 2011), and the german standard for the same purpose, DIN 4844-2 (*DIN 4844-2 (2011)* 2010). If the value is 'Meteorological', a simplified house and a cyclone are depicted. If the value is 'Seismic', a simplified house is depicted with a crack inside. The house stands on ground that has a chasm. If the value is 'Social', three persons are depicted, where one person is lying on the ground. One of the two standing persons is kicking the lying person.

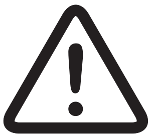


Figure A.57.: HazardImageUncategorized

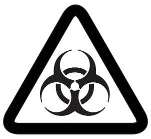


Figure A.58.: HazardImageBiological

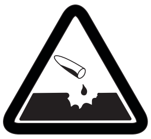


Figure A.59.: HazardImageChemical

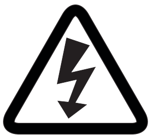


Figure A.60.: HazardImageElectrical

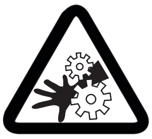


Figure A.61.: HazardImageMechanical

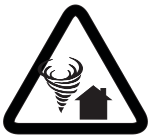


Figure A.62.: HazardImageMeteorological

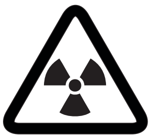


Figure A.63.: HazardImageRadiological

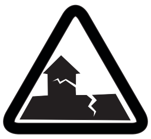


Figure A.64.: HazardImageSeismic

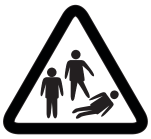


Figure A.65.: HazardImageSocial

A.4.41. HazardType (Enumeration)

Categorized Class

- Hazard

Enumeration Literals

- Seismical
- Meteorological
- Biological
- Chemical
- Radiological
- Mechanical
- Electrical
- Social
- Uncategorized

A.4.42. Idea (Class)

Generalizations

- URMLModelEntity

Description An inspiration or thought that may give rise to (i.e. motivates) a stakeholder request. An idea is also mentioned by a stakeholder and thus has to be differentiated from a request or a goal.

Attributes

- source: URI

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)

Concrete Syntax and Mapping The notation for Idea is an entity notation. It uses the name attribute of the Idea meta-class to label its instances. The other attributes are currently not reflected in the notation.

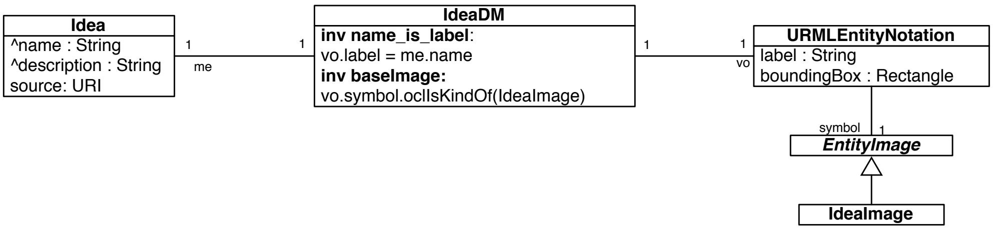


Figure A.66.: Idea Notation

Images The notation uses exactly one image, IdeaImage. It depicts a thought bubble with a shining light bulb inside.

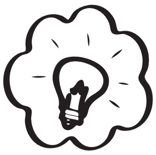


Figure A.67.: IdeaImage

A.4.43. InformationFlow (Relationship)

Enables modeling the flow of entity objects between processes.

Generalizations

- URMLModelRelationship

Description This relationship connects an entity object with a process. Whether it is going into the process (input) or coming out of it (output) should be understandable from the context of the model elements and the value of the type attribute. A process may create, delete, modify, or use an entity object. If a single process is the only user of an entity object, the relationship may be left uncategorized.

Attributes

- type: InformationFlowType

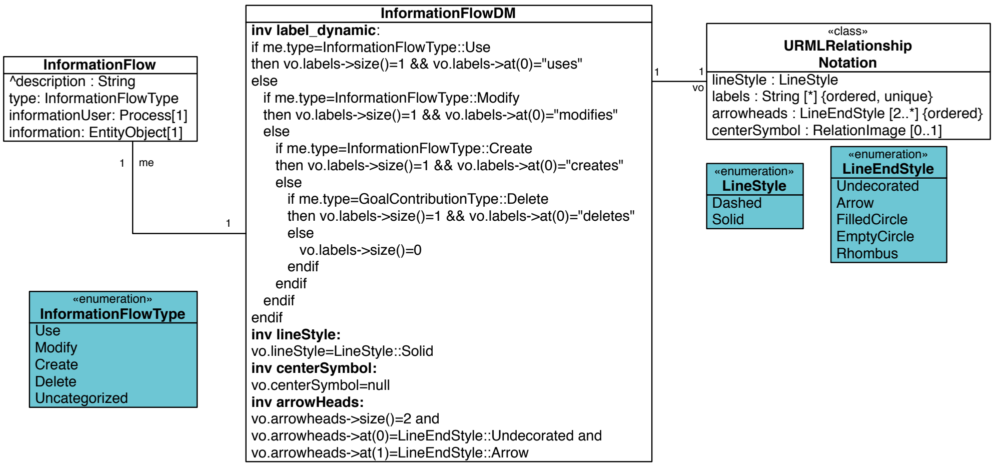
Role Names and Cardinalities

informationUser: Process[1]

information: EntityObject[1]

Template Sentence “Process works on EntityObject”

Concrete Syntax and Mapping The notation for InformationFlow is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at EntityObject, and a dynamic text label determined by the value of the type attribute.



```
inv label_dynamic:
if me.type=InformationFlowType::Use
then vo.labels->size()=1 && vo.labels->at(0)="uses"
else
  if me.type=InformationFlowType::Modify
  then vo.labels->size()=1 && vo.labels->at(0)="modifies"
  else
    if me.type=InformationFlowType::Create
    then vo.labels->size()=1 && vo.labels->at(0)="creates"
    else
      if me.type=GoalContributionType::Delete
      then vo.labels->size()=1 && vo.labels->at(0)="deletes"
      else
        vo.labels->size()=0
      endif
    endif
  endif
endif
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle::Undecorated and
vo.arrowheads->at(1)=LineEndStyle::Arrow
```

Figure A.68.: InformationFlow Notation

Images No special imagery.

A.4.44. InformationFlowType (Enumeration)

Categorized Class

- InformationFlow

Enumeration Literals

- Create
- Delete

- Modify
- Uncategorized
- Use

Description

Create - The process creates the object. Only one process may create an object.

Delete - The process deletes the object. Only one process may delete an object.

Modify - The process modifies the object.

Uncategorized - The process is the single user of the object does all types of interaction with it.

Use - The process uses the object but does not modify.

A.4.45. Inheritance (Relationship)

Expresses generalization/specialization between model entities.

Generalizations

- URMLModelRelationship

Description The semantics of the Inheritance relationship is similar to the generalization relationship of UML. If an instance of a URMLEntity subclass inherits from a more general one, it inherits all its properties. The more specific entity may have additional properties that should not conflict with what is already inherited.

Attributes

- No additional attributes.

Role Names and Cardinalities

superURMLModelEntity: URMLModelEntity[1]

subURMLModelEntity: URMLModelEntity[1]

Template Sentence “URMLModelEntity inherits from URMLModelEntity”

Concrete Syntax and Mapping The notation for Inheritance is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at Idea, and the static label text ’inherits’.

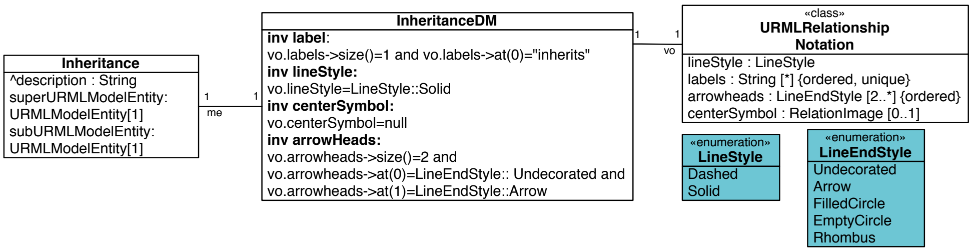


Figure A.69.: Inheritance Notation

Images No special imagery.

A.4.46. Mention (Relationship)

Shows which stakeholder mentioned an idea.

Generalizations

- URMLModelRelationship

Description For every idea recorded in the model, it should be known which stakeholder mentioned the idea. This helps analyzing which requests are motivated by that idea.

Attributes

- No additional attributes.

Role Names and Cardinalities

mentioningStakeholder: Stakeholder[1]

mentionedIdea: Idea[1]

Template Sentence “Stakeholder mentions Idea”

Concrete Syntax and Mapping The notation for Mention is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at Idea, and the static label text 'mentions'.

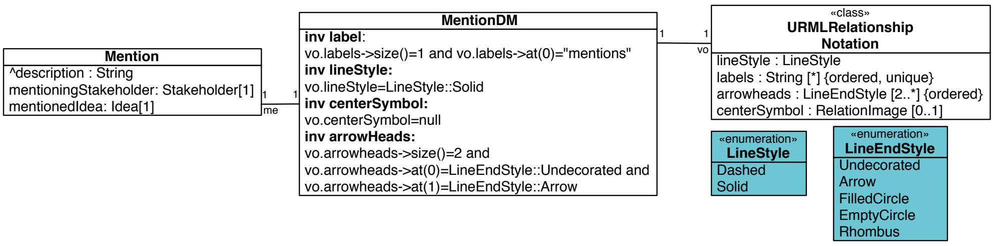

Mention
^description : String
mentioningStakeholder: Stakeholder[1]
mentionedIdea: Idea[1]
1
1
me
MentionDM
inv label:
vo.labels->size()=1 and vo.labels->at(0)="mentions"
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle::Undecorated and
vo.arrowheads->at(1)=LineEndStyle::Arrow
1
1
vo
«class»
URMLRelationship Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.70.: Mention Notation

Images No special imagery.

A.4.47. Mitigation (Abstract Relationship)

Generalizations

- URMLModelRelationship

Description mitigation is an abstract relationship that expresses that a danger is being dealt with. This relationship can not be used on a diagram. Modelers have to use either a requirement mitigation (i.e. the danger is mitigated by a requirement that describes how the system should take precautions, circumvent the danger, or reduce the likeliness of an incident) or a procedural mitigation (i.e. the precautions are not built into the system but are realized by procedures that come along with the system, e.g. a handbook or staff training).

Attributes

- No additional attributes.

Role Names and Cardinalities

mitigatedDanger: Danger[1]

protectedHarmedElement: HarmedElement[1]

Images MitigationImage depicts a shield.

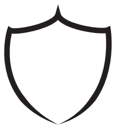


Figure A.71.: MitigationImage

A.4.48. Motivation (Relationship)

Shows which idea motivated a request.

Generalizations

- URMLModelRelationship

Description Some knowledge the stakeholder had in mind, which was externalized in the form of an idea to the model, can be the motivation of a request. This relationship maps the idea to the request it motivated. Ideas that are not connected to a request might be a hint to future requests.

Attributes

- No additional attributes.

Role Names and Cardinalities

motivatingIdea: Idea[1]

motivatedRequest: Request[1]

Template Sentence “Idea motivates Request”

Concrete Syntax and Mapping The notation for Motivation is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at Request, and the static label text ’motivates’.

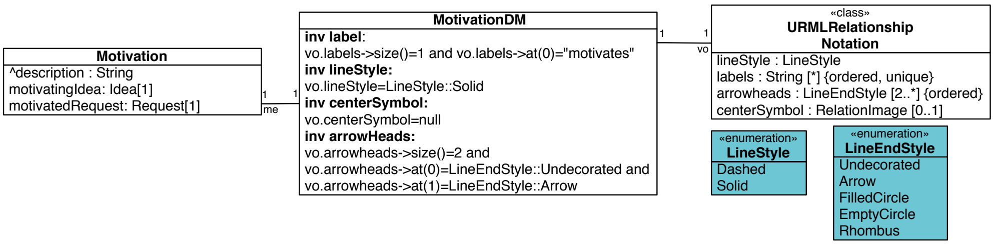

Motivation
^description : String
motivatingIdea: Idea[1]
motivatedRequest: Request[1]
1
1
me
MotivationDM
inv label:
vo.labels->size()=1 and vo.labels->at(0)="motivates"
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle::Undecorated and
vo.arrowheads->at(1)=LineEndStyle::Arrow
1
1
vo
«class»
URMLRelationship
Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.72.: Motivation Notation

Images No special imagery.

A.4.49. Procedure (Class)

Something outside the system under discussion that tells users how to safely use the system.

Generalizations

- URMLModelEntity

Description A procedure is not built into the system but into the environment of system. It prescribes certain constraints or workflows how to use the system. For example it can be a warning in the user manual containing safety instructions, guidelines in general, or staff training.

Attributes

- No additional attributes.

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)

Concrete Syntax and Mapping The notation for Procedure is an entity notation. It uses the name attribute of the Procedure meta-class to label its instances. The description attribute is currently not reflected in the notation.

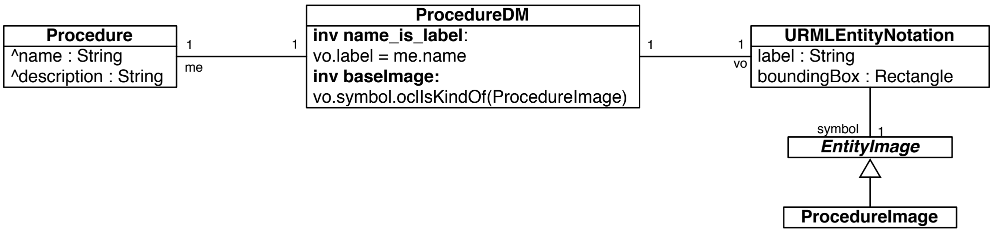


Figure A.73.: Procedure Notation

Images The notation uses exactly one image, ProcedureImage. It depicts an open book, with two symbols on the open pages. On the left page, a wrench is depicted and on the right page a gear-wheel.

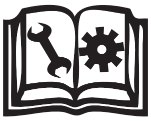


Figure A.74.: ProcedureImage

Suggestions to implementors If the word procedure is a reserved word in your environment, rename the class to “MitigatingProcedure”.

A.4.50. ProceduralMitigation (Relationship)

Protects a harmed element from danger through a procedure.

Generalizations

- Mitigation

Description A procedural mitigation is a mitigation relationship that expresses that a danger is mitigated by a procedure. It signifies that a danger is dealt with by precautions external to the system under discussion.

Attributes

- No additional attributes.

Inherited Role Names and Cardinalities

mitigatedDanger (from Mitigation)

protectedHarmedElement (from Mitigation)

Role Names and Cardinalities

mitigatingProcedure: Procedure[1..*]

Template Sentence “Procedure protects HarmedElement from Danger”

Concrete Syntax and Mapping The notation for ProceduralMitigation is a relationship notation. It has a solid line, no arrowheads at any line end three different static label texts ’protect’, ’from’, ’with’. It has the center symbol MitigationImage.

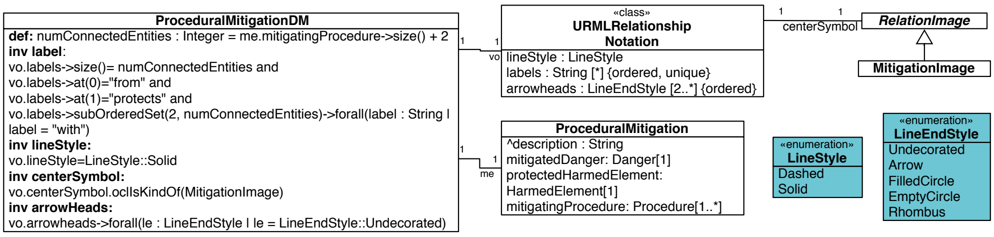

ProceduralMitigationDM
def: numConnectedEntities : Integer = me.mitigatingProcedure->size() + 2
inv label:
vo.labels->size()= numConnectedEntities and
vo.labels->at(0)="from" and
vo.labels->at(1)="protects" and
vo.labels->subOrderedSet(2, numConnectedEntities)->forall(label : String l
label = "with")
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol.oclIsKindOf(MitigationImage)
inv arrowHeads:
vo.arrowheads->forall(le : LineEndStyle l le = LineEndStyle::Undecorated)
1
1
vo
«class»
URMLRelationship
Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
1
1
centerSymbol
RelationImage
MitigationImage
1
1
me
ProceduralMitigation
^description : String
mitigatedDanger: Danger[1]
protectedHarmedElement:
HarmedElement[1]
mitigatingProcedure: Procedure[1..*]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.75.: ProceduralMitigation Notation

Images MitigationImage is presented in Section A.4.47.

A.4.51. Process (Abstract Class)

Generalizations

- URMLModelEntity

Description A process is an abstract concept that describes the system with regards to its usage. When specialized as EnvironmentProcess, it describes the environment in which the system is used (i.e. the business processes of the customer). When specialized as UseCase, it describes the functions that the system offers to its users (i.e. actors). A process in general describes the necessary steps to achieve something. It also describes the objects used in these processes; such as objects the users interact with (i.e. boundary objects),

objects that represent internal state (i.e. entity objects), and internal agents that control the processes (i.e. service providers). A process that is marked being atomic cannot be subdivided further into processes. Business processes can be modeled by setting the businessProcess attribute to true. A process has pre- and post-conditions. It can be related to other processes via “include” and “extend” relationships (similar to UML). A process can trigger a danger and can be vulnerable to danger. The attribute underConstruction is used for model reviews, when the reviewer needs to know whether the modeler is still working on the details of a particular process. This means that there are still subprocesses to be modeled. The attributes atomic and underConstruction can not be true at the same time, as an atomic process has no subprocesses and thus there is no details to be worked on.

Attributes

- atomic: Boolean
- businessProcess: Boolean
- underConstruction: Boolean
- preCondition: String
- postCondition: String

Inherited Attributes

- name (from URMLModelEntity)
- description (from URMLModelElement)

Constraints

- atomic and underConstruction cannot be true at the same time

A.4.52. ProcessConstraint (Relationship)

Connects a process with a quality requirement that constrains the process.

Generalizations

- URMLModelRelationship

Description A process may be required to execute within a certain time slot, or only with a restricted set of resources. These are just two examples of how quality requirements can constrain (and thus detail) how a process can be executed. Every type of quality requirement can formulate a constraint to a process.

Attributes

- No additional attributes.

Role Names and Cardinalities

constrainedProcess: Process[1]

constrainingQualityRequirement: QualityRequirement[1]

Template Sentence “QualityRequirement constrains Process”

Concrete Syntax and Mapping The notation for ProcessConstraint is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at Process, and the static label text ’constrains’.

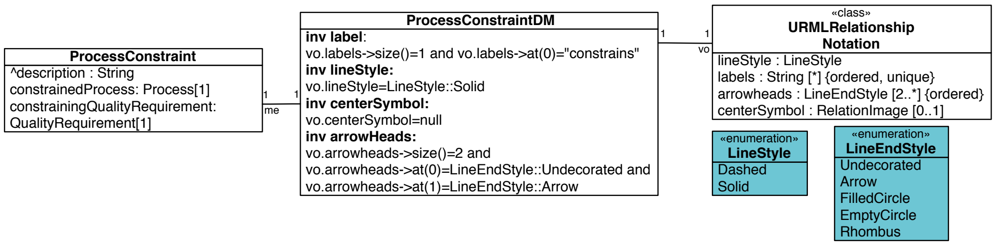

ProcessConstraint
^description : String
constrainedProcess: Process[1]
constrainingQualityRequirement:
QualityRequirement[1]
1
1
me
ProcessConstraintDM
inv label:
vo.labels->size()=1 and vo.labels->at(0)="constrains"
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle::Undecorated and
vo.arrowheads->at(1)=LineEndStyle::Arrow
1
1
vo
«class»
URMLRelationship
Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.76.: ProcessConstraint Notation

Images No special imagery.

A.4.53. ProcessEnabling (Relationship)

Shows which environment process is enabled by a feature of the system under construction.

Generalizations

- URMLModelRelationship

Description A system enables certain processes of the environment with its features (e.g. the photocopier enabled selling copies in copy shops). This is especially of interest for a system under construction, as its usability and profitability has to be defended. This relationship allows to link environment processes with features and thus helps with this task.

Attributes

- No additional attributes.

Role Names and Cardinalities

enabledProcess: EnvironmentProcess[1]

enablingFeature: Feature[1]

Template Sentence “Feature enables EnvironmentProcess”

Concrete Syntax and Mapping The notation for ProcessEnabling is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at EnvironmentProcess, and the static label text ’enables’.

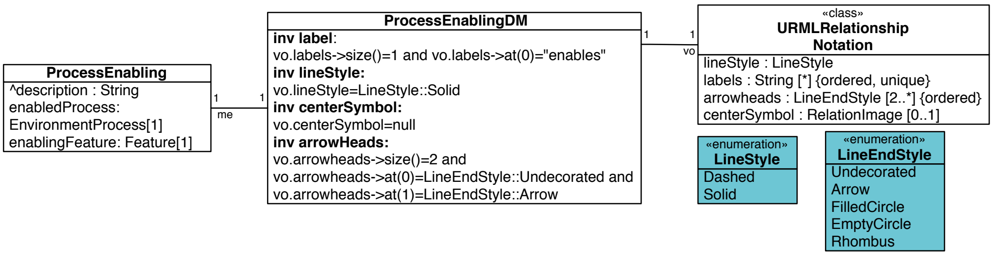

ProcessEnabling
^description : String
enabledProcess:
EnvironmentProcess[1]
enablingFeature: Feature[1]
1
me
1
ProcessEnablingDM
inv label:
vo.labels->size()=1 and vo.labels->at(0)="enables"
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle::Undecorated and
vo.arrowheads->at(1)=LineEndStyle::Arrow
1
1
vo
«class»
URMLRelationship
Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.77.: ProcessEnabling Notation

Images No special imagery.

A.4.54. ProcessExtension (Relationship)

Lets a process extend another.

Generalizations

- URMLModelRelationship

Description Process extension is a well-known concept incorporated from the UML (which incorporated Jacobson's use case approach). Process extension expresses optional or exceptional behavior of process. This behavior is specified by the extending process.

Attributes

- No additional attributes.

Role Names and Cardinalities

extendingProcess: Process[1]

extendedProcess: Process[1]

Template Sentence "Process extends Process"

Concrete Syntax and Mapping The notation for ProcessExtension is a relationship notation. It has a dashed line, no center symbol, an arrow-style arrowhead at the end of line ending at extendedProcess, and the static label text '≪extend≫'.

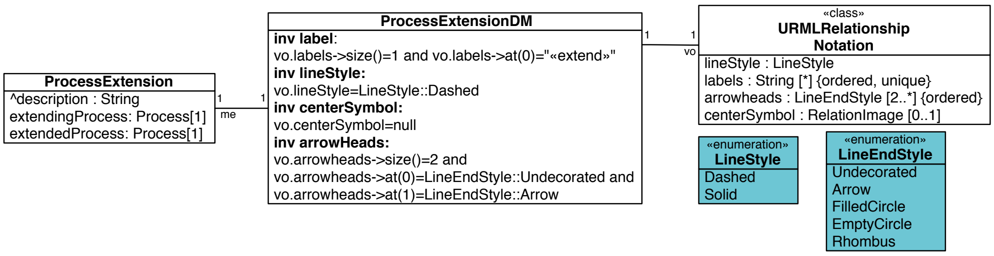

ProcessExtension
^description : String
extendingProcess: Process[1]
extendedProcess: Process[1]
1
me
1
ProcessExtensionDM
inv label:
vo.labels->size()=1 and vo.labels->at(0)="«extend»"
inv lineStyle:
vo.lineStyle=LineStyle::Dashed
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle::Undecorated and
vo.arrowheads->at(1)=LineEndStyle::Arrow
1
1
vo
«class»
URMLRelationship
Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.78.: ProcessExtension Notation

Images No special imagery.

A.4.55. ProcessInclusion (Relationship)

Lets a process include another.

Generalizations

- URMLModelRelationship

Description Process inclusion also is a well-known concept incorporated from the UML (see ProcessExtension). Process inclusion allows the decomposition of processes. So a complex process can be assembled from simpler subprocesses. How the subprocesses interact with each other can be modeled through the precedence relationship (see ProcessPrecedence). The URML does not support partial decomposition, so the behavior of an including process is fully specified by its included and extending processes.

Attributes

- No additional attributes.

Role Names and Cardinalities

includingProcess: Process[1]

includedProcess: Process[1]

Template Sentence “Process includes Process”

Concrete Syntax and Mapping The notation for ProcessInclusion is a relationship notation. It has a dashed line, no center symbol, an arrow-style arrowhead at the end of line ending at includedProcess, and the static label text '≪include≫'.

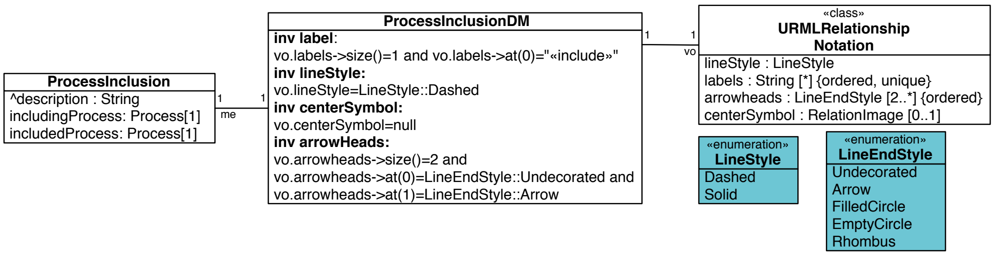

ProcessInclusion
^description : String
includingProcess: Process[1]
includedProcess: Process[1]
1
1
me
ProcessInclusionDM
inv label:
vo.labels->size()=1 and vo.labels->at(0)="«include»"
inv lineStyle:
vo.lineStyle=LineStyle::Dashed
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle::Undecorated and
vo.arrowheads->at(1)=LineEndStyle::Arrow
1
1
vo
«class»
URMLRelationship
Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.79.: ProcessProcessInclusion Notation

Images No special imagery.

A.4.56. ProcessPrecedence (Relationship)

Shows the order of processes.

Generalizations

- URMLModelRelationship

Description This relationships enables a partial ordering of processes. While the URML is not as powerful regarding control flow modeling as the BPMN or UML Activity diagrams, it offers modeling a sequence of processes through this relationship. Processes can also occur in parallel. Decision gates are not supported. Detailed control flow should be left to the designer of the system under construction. The URML attempts to offer a basic set of control flow modeling features because precedence might be important for understanding certain requirements, whose formulation would imply that one process executes before another. The precedence relationship is also useful in combination with the information flow relationship (see InformationFlow). The modeler can see how information can be routed through the system.

Attributes

- No additional attributes.

Role Names and Cardinalities

precedingProcess: Process[1]

precededProcess: Process[1]

Template Sentence "Process precedes Process"

Concrete Syntax and Mapping The notation for ProcessPrecedence is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at precededProcess, and the static label text 'precedes'.

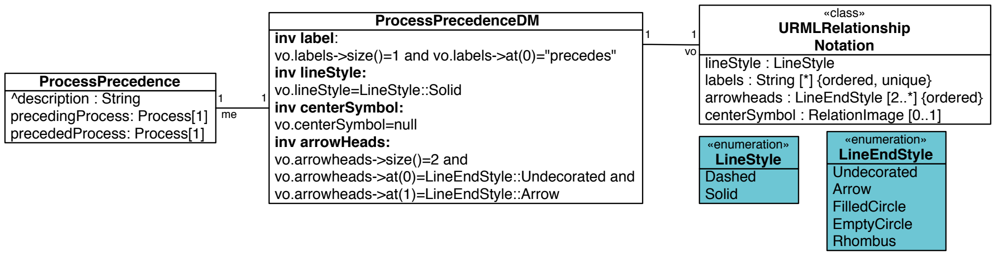

ProcessPrecedence
^description : String
precedingProcess: Process[1]
precededProcess: Process[1]
1
1
me
ProcessPrecedenceDM
inv label:
vo.labels->size()=1 and vo.labels->at(0)="precedes"
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle::Undecorated and
vo.arrowheads->at(1)=LineEndStyle::Arrow
1
1
vo
«class»
URMLRelationship Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.80.: ProcessPrecedence Notation

Images No special imagery.

Known Issues A URML user reported that a differentiation of two kinds of precedence is needed: First, simple precedence where one process is taking place before the other. Second, one process causes the other to start, i.e. triggers the other process. This is already implemented in the EA URML Add-In, but not reflected in the abstract syntax of URML.

A.4.57. ProcessRequirement (Relationship)

Facilitates modeling of prerequisite functional requirements.

Generalizations

- URMLModelRelationship

Description A given process may only be exectutable if a given functional requirement is realized. This relationship provides a more specific viewpoint as the FeatureDescriptionUseCase relationship. In the latter, a UseCase instance is detailed by Feature instances. With ProcessRequirement, the modeler can highlight those FunctionalRequirement instances of such features with mandatory importance to the described Process instance.

Attributes

- No additional attributes.

Role Names and Cardinalities

detailedProcess: Process[1]

detailingFunctionalRequirement: FunctionalRequirement[1]

Template Sentence “Process requires FunctionalRequirement”

Concrete Syntax and Mapping The notation for ProcessRequirement is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of the line ending at detailingFunctionalRequirement, and the static label text ’requires’.

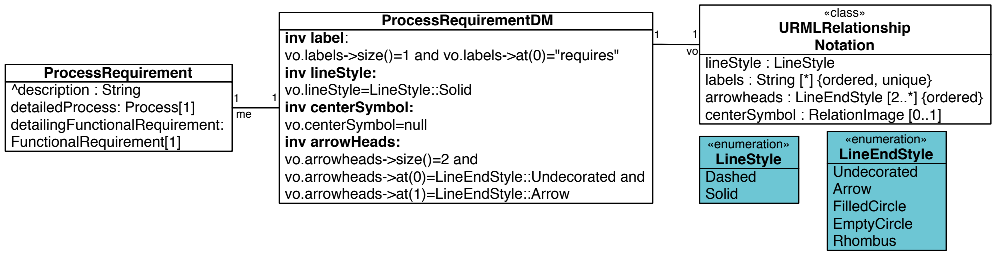

ProcessRequirement
^description : String
detailedProcess: Process[1]
detailingFunctionalRequirement:
FunctionalRequirement[1]
1
1
me
ProcessRequirementDM
inv label:
vo.labels->size()=1 and vo.labels->at(0)="requires"
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle::Undecorated and
vo.arrowheads->at(1)=LineEndStyle::Arrow
1
1
vo
«class»
URMLRelationship
Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.81.: ProcessRequirement Notation

Images No special imagery.

A.4.58. Product (Class)

Generalizations

- URMLModelEntity

Description Something that is marketed or sold. It may be a physical entity that is manufactured or may be a set of services. Through the 'upcoming' attribute, it can be indicated whether the product already exists or if it is a future product

Attributes

- upcoming: Boolean

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)

Concrete Syntax and Mapping The notation for Product is an entity notation. It uses the name attribute of the Product meta-class to label its instances. The description attribute is currently not reflected in the notation. The upcoming attribute switches between the ProductImagePresent and ProductImageFuture images.

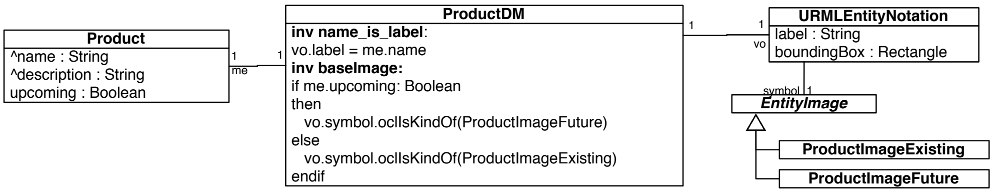


Figure A.82.: Product Notation

Images The overall icon consists of a base icon that is filled with different content depending on the value of the upcoming attribute. The base icon is a three-dimensional box. If upcoming is set to false, the front side has square shape, if it is set to true, a cloud is on the front side.

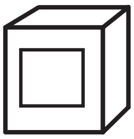


Figure A.83.: Product-Image-Existing

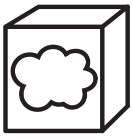


Figure A.84.: Product-Image-Future

A.4.59. ProductLine (Class)

Generalizations

- URMLModelEntity

Description A product line defines a set of products sharing a common managed set of features that satisfy the specific needs of a selected market. A product line points to the root of a feature tree. The feature tree of the product line describes all possible current and future product features that are used to define products from the product line.

Attributes

- No additional attributes.

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)

Concrete Syntax and Mapping The notation for ProductLine is an entity notation. It uses the name attribute of the ProductLine meta-class to label its instances. The description attribute is currently not reflected in the notation.

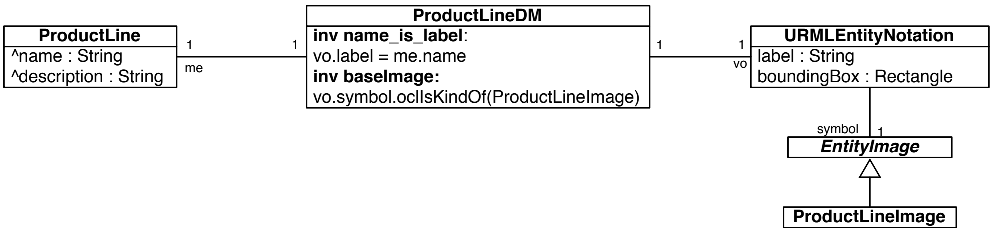


Figure A.85.: ProductLine Notation

Images The notation uses exactly one image, ProductLineImage. It depicts a large box on which three smaller boxes are standing, one of them being taller than the other two.

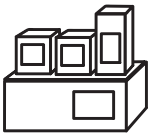


Figure A.86.: ProductLineImage

A.4.60. ProductLineAggregation (Relationship)

Shows the parent of a product line in a multi-leveled product line.

Generalizations

- URMLModelRelationship

Description Product lines may contain other product lines (e.g. in car manufacturing, a company may have different brands, each forming its own product line, but still member of the company-wide car product line.). This relationship shows to which product line a certain product line belongs to.

Attributes

- No additional attributes.

Role Names and Cardinalities

containingProductLine: ProductLine[1]

containedProductLine: ProductLine[1]

Template Sentence “ProductLine contains ProductLine”

Concrete Syntax and Mapping The notation for ProductLineAggregation is a relationship notation. It has a solid line, no center symbol, an rhombus-style arrowhead at the end of line ending at containingProductLine, and no label.

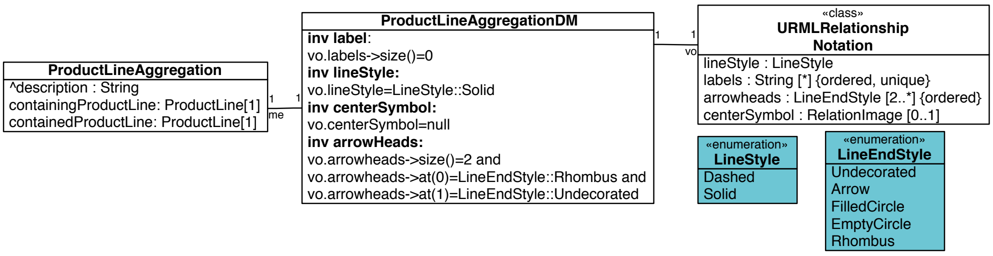

ProductLineAggregation
^description : String
containingProductLine: ProductLine[1]
containedProductLine: ProductLine[1]
1
1
me
ProductLineAggregationDM
inv label:
vo.labels->size()=0
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle::Rhombus and
vo.arrowheads->at(1)=LineEndStyle::Undecorated
1
1
vo
«class»
URMLRelationship
Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.87.: ProductLineAggregation Notation

Images No special imagery.

A.4.61. ProductLineContainment (Relationship)

Shows which product line the product belongs to.

Generalizations

- URMLModelRelationship

Description This relationship allows to show which product line a product was created from. A product can only be part of one product line (implicit participation through the product line aggregation relationship not counting).

Attributes

- No additional attributes.

Role Names and Cardinalities

containingProductLine: ProductLine[1]

containedProduct: Product[1]

Template Sentence "Product is part of ProductLine"

Concrete Syntax and Mapping The notation for ProductLineContainment is a relationship notation. It has a solid line, no center symbol, an rhombus-style arrowhead at the end of line ending at containingProductLine, and no label.

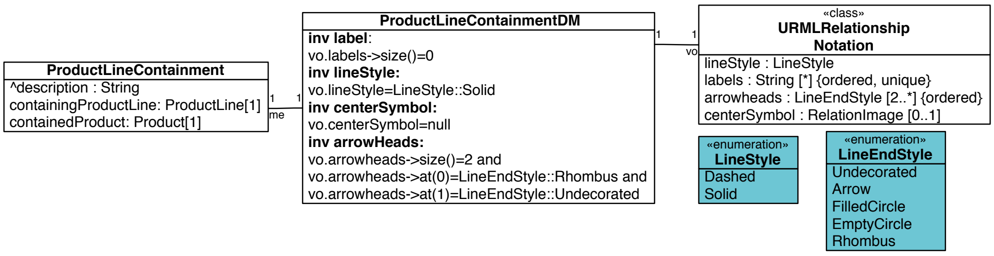

ProductLineContainment
^description : String
containingProductLine: ProductLine[1]
containedProduct: Product[1]
1
1
me
ProductLineContainmentDM
inv label:
vo.labels->size()=0
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle::Rhombus and
vo.arrowheads->at(1)=LineEndStyle::Undecorated
1
1
vo
«class»
URMLRelationship Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.88.: ProductLineContainment Notation

Images No special imagery.

A.4.62. ProductSuite (Class)

Generalizations

- URMLModelEntity

Description A product suite is group of products that complement each other and can be marketed as a complete set, e.g., the Microsoft Office Suite.

Attributes

- No additional attributes.

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)

Concrete Syntax and Mapping The notation for ProductSuite is an entity notation. It uses the name attribute of the ProductSuite meta-class to label its instances. The description attribute is currently not reflected in the notation.

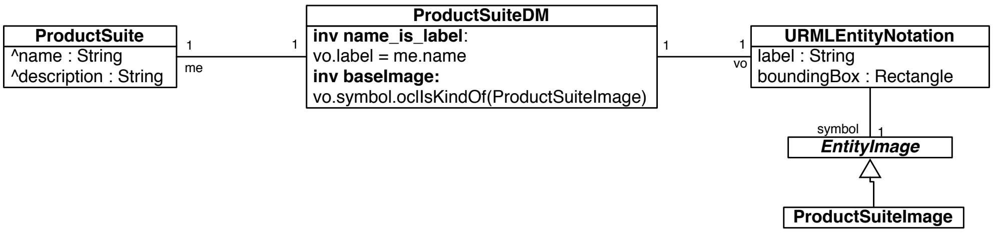


Figure A.89.: ProductSuite Notation

Images The notation uses exactly one image, ProductSuiteImage. It depicts three three-dimensional boxes, each with a tool inside. One box is standing on the two others.

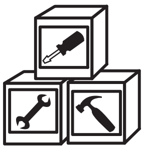


Figure A.90.: ProductSuiteImage

A.4.63. ProductSuiteContainment (Relationship)

Shows if a product is member of a product suite.

Generalizations

- URMLModelRelationship

Description This relationship allows indicating the membership of a product in product suites. This means that the product is or will be sold in a bundle with the other products of the product suite.

Attributes

- No additional attributes.

Role Names and Cardinalities

containingProductSuite: ProductSuite[1]

productSuiteMember: Product[1]

Template Sentence "Product is member of ProductSuite"

Concrete Syntax and Mapping The notation for ProductSuiteContainment is a relationship notation. It has a solid line, no center symbol, an rhombus-style arrowhead at the end of line ending at containingProductSuite, and no label.

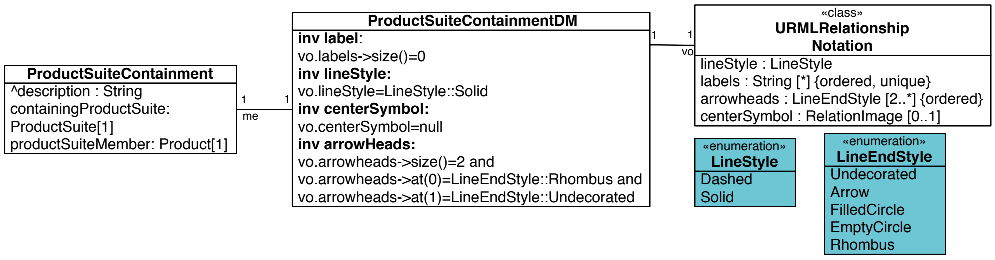

ProductSuiteContainment
^description : String
containingProductSuite: ProductSuite[1]
productSuiteMember: Product[1]
1
1
me
ProductSuiteContainmentDM
inv label:
vo.labels->size()=0
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle::Rhombus and
vo.arrowheads->at(1)=LineEndStyle::Undecorated
1
1
vo
«class»
URMLRelationship Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.91.: ProductSuiteContainment Notation

Images No special imagery.

A.4.64. QualityRequirement (Class)

A quality requirement is "a requirement that pertains to a quality concern that is not covered by functional requirements" (Glinz 2014).

Generalizations

- Requirement

Description A quality requirement is a class of requirements describing a constraint on the operation of a product, process or service, e.g. performance, environmental, quality. The type of the quality requirement is specified by an enumeration type, inspired by ISO 9126 categories. A quality requirement may constrain the system or its environment: it can constrain a process (i.e. it constrains either environment processes or use cases). Quality requirements provides details quality concerns of system features.

There is one exceptional category of quality requirements that has been incorporated upon the request of a URML user: the QualityRequirementType ProjectExecution. It represents requirements regarding the quality of the development process. Such requirements are usually modeled as constraints, and therefore this QualityRequirement type should be considered a hack that should be refactored in future versions of URML, where it could be moved to a dedicated abstraction that models constraints.

Constraints If a quality requirement is about project execution, it should neither constrain any use case nor be part of a feature.
inv project_execution_qr_not_referring_system:
type == QualityRequirementType.ProjectExecution implies
featureConstraints.isEmpty() and
forall (pr:processRequirements | !pr.process.oclIsKindOf(UseCase))

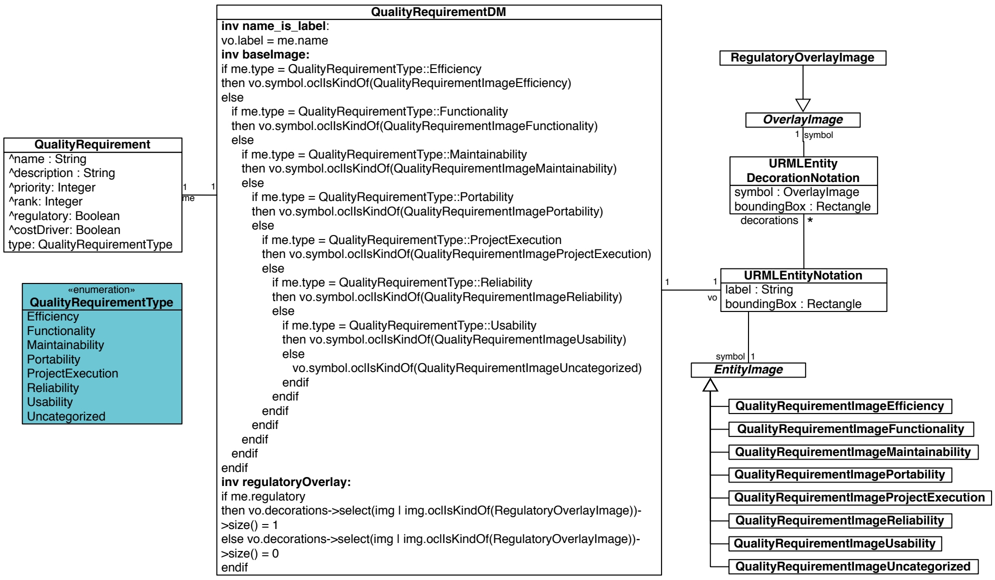
Attributes

- type: QualityRequirementType

Inherited Attributes

- costDriver (from Requirement)
- description (from URMLModelElement)
- name (from URMLModelElement)
- priority (from Requirement)
- rank (from Requirement)
- regulatory (from Requirement)

Concrete Syntax and Mapping The notation for QualityRequirement is an entity notation. It uses the name attribute of the QualityRequirement metaclass to label its instances. Depending on the regulatory attribute, an overlay image is applied to to base image. If the regulatory attribute is true, exactly one instance of RegulatoryOverlayImage is applied to the base QualityRequirementImage*, if false there must be no decoration of that kind. The value of the type attribute determines which image is used for the QualityRequirement instance. The other attributes of FunctionalRequirement are currently not reflected in the notation.



```
inv name_is_label:
vo.label = me.name
inv baseImage:
if me.type = QualityRequirementType::Efficiency
then vo.symbol.oclIsKindOf(QualityRequirementImageEfficiency)
else
  if me.type = QualityRequirementType::Functionality
  then vo.symbol.oclIsKindOf(QualityRequirementImageFunctionality)
  else
    if me.type = QualityRequirementType::Maintainability
    then vo.symbol.oclIsKindOf(QualityRequirementImageMaintainability)
    else
      if me.type = QualityRequirementType::Portability
      then vo.symbol.oclIsKindOf(QualityRequirementImagePortability)
      else
        if me.type = QualityRequirementType::ProjectExecution
        then vo.symbol.oclIsKindOf(QualityRequirementImageProjectExecution)
        else
          if me.type = QualityRequirementType::Reliability
          then vo.symbol.oclIsKindOf(QualityRequirementImageReliability)
          else
            if me.type = QualityRequirementType::Usability
            then vo.symbol.oclIsKindOf(QualityRequirementImageUsability)
            else
              vo.symbol.oclIsKindOf(QualityRequirementImageUncategorized)
            endif
          endif
        endif
      endif
    endif
  endif
endif
inv regulatoryOverlay:
if me.regulatory
then vo.decorations->select(img | img.oclIsKindOf(RegulatoryOverlayImage))-
>size() = 1
else vo.decorations->select(img | img.oclIsKindOf(RegulatoryOverlayImage))-
>size() = 0
endif
```

Figure A.92.: QualityRequirement Notation

Images The final symbol representing instances of QualityRequirement consists of a base image that is optionally decorated with an overlay image. The base image depends on the value of the type attribute. The base image is decorated with an overlay image if the value of the regulatory attribute is set to true. The base image always depicts a diamond. The modification depending on the type is an uppercase letter which is the first letter of the type literal. In the case of the project execution type, a deviation from this rule exists as the letter 'P' was already taken by the portability type. In this case, two crossed tools (hammer and wrench) are depicted "inside" the diamond. The regulatory overlay depicts a police officer.

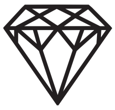


Figure A.93.: QualityRequirementImage Uncategorized

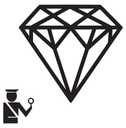


Figure A.94.: Uncategorized, with RegulatoryOverlayImage

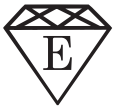


Figure A.95.: QualityRequirementImage Efficiency

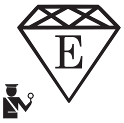


Figure A.96.: Efficiency, with RegulatoryOverlayImage

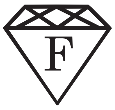


Figure A.97.: QualityRequirementImage Functionality

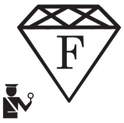


Figure A.98.: Functionality, with RegulatoryOverlayImage

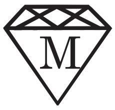


Figure A.99.: QualityRequirementImage Maintainability

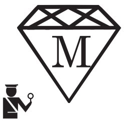


Figure A.100.: Maintainability with RegulatoryOverlayImage

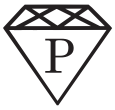


Figure A.101.: QualityRequirementImage Portability

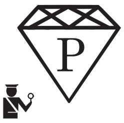


Figure A.102.: Portability with RegulatoryOverlayImage

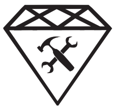


Figure A.103.: QualityRequirementImage ProjectExecution

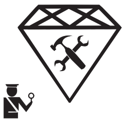


Figure A.104.: ProjectExecution, with RegulatoryOverlayImage

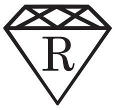


Figure A.105.: QualityRequirementImage Reliability

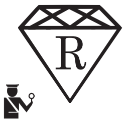


Figure A.106.: Reliability with RegulatoryOverlayImage

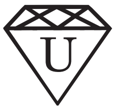


Figure A.107.: QualityRequirementImage Usability

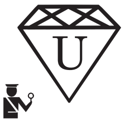


Figure A.108.: Usability with RegulatoryOverlayImage

A.4.65. QualityRequirementType (Enumeration)

Categorized Class

- QualityRequirement

Enumeration Literals

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability
- Project Execution
- Uncategorized

Description Most of the enumeration literals where adapted from ISO 9126. Therefore, the ISO's definitions are usually directly cited from (ISO 2001). Where appropriate, additional explanations were added.

Functionality: Incorporated from ISO 9126: "Functionality quality requirements target the functional suitability of the system. They provide a metric to determine whether the system is appropriate to its users." Entails suitability, accuracy, and interoperability. The ISO 9126 standard also includes security, which is addressed by mitigating requirements in the URML.

Reliability: Incorporated from ISO 9126: "The capability of the [...] product to maintain a specified level of performance when used under specified conditions." Entails maturity, fault tolerance, and recoverability.

Usability: Incorporated from ISO 9126: "The capability of the [...] product to be understood, learned, used and attractive to the user, when used under specified conditions." Entails understandability, learnability, operability, and attractiveness.

Efficiency: Incorporated from ISO 9126: “The capability of the [...] product to provide appropriate performance, relative to the amount of resources used, under stated conditions.” Entails time behavior and resource utilization.

Maintainability: Incorporated from ISO 9126: “The capability of the [...] product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.” Entails analyzability, changeability, stability, and testability.

Portability: Incorporated from ISO 9126: “The capability of the [...] product to be transferred from one environment to another.” Entails coexistence and replaceability.

ProjectExecution: Quality requirement that constrains the process of creating the system under discussion. All others are constraints to the system under discussion itself.

Uncategorized: The category of the quality requirement is yet to be determined.

A.4.66. Request (Class)

Generalizations

- URMLModelEntity

Description Asking for something to be part of or a constraint on a system. Requests, which are wishes and suggestions expressed by stakeholders. Requests can also yield requirements.

Attributes

- No additional attributes.

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)

Concrete Syntax and Mapping The notation for Request is an entity notation. It uses the name attribute of the Request meta-class to label its instances. The description attribute is currently not reflected in the notation.

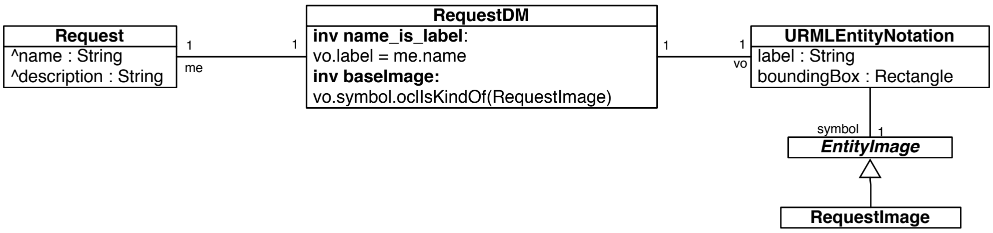


Figure A.109.: Request Notation

Images The notation uses exactly one image, RequestImage. It depicts a person raising a hand with a speech bubble that has a question mark inside.

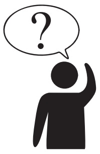


Figure A.110.: RequestImage

A.4.67. RequestStatement (Relationship)

Shows which stakeholder expressed a request.

Generalizations

- URMLModelRelationship

Description This relationships allows the mapping of stakeholders to requests. Thus it can be analyzed who was the source of a request or if multiple stakeholders uttered a similar request.

Attributes

- No additional attributes.

Role Names and Cardinalities

issuingStakeholder: Stakeholder[1]

statedRequest: Request[1]

Template Sentence “Stakeholder expresses Request”

Concrete Syntax and Mapping The notation for RequestStatement is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at Request, and the static label text ’expresses’.

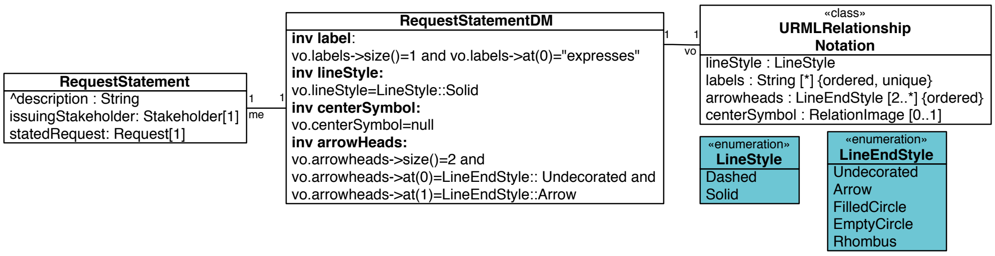

RequestStatement
^description : String
issuingStakeholder: Stakeholder[1]
statedRequest: Request[1]
1
1
me
RequestStatementDM
inv label:
vo.labels->size()=1 and vo.labels->at(0)="expresses"
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle:: Undecorated and
vo.arrowheads->at(1)=LineEndStyle::Arrow
1
1
vo
«class»
URMLRelationship Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.111.: RequestStatement Notation

Images No special imagery.

A.4.68. RequestCause (Relationship)

Allows analysis of the goal behind a request.

Generalizations

- URMLModelRelationship

Description Requests do not come out of the void. While they may be motivated by general knowledge (i.e. ideas; see motivation relationship) they can also be there because a stakeholder has a certain goal in mind. If this can be detected during elicitation, this relationship can be used to indicate which goal is behind a stakeholder request.

Attributes

- No additional attributes.

Role Names and Cardinalities

backdropGoal: Goal[1]

resultingRequest: Request[1]

Template Sentence “Goal results in Request”

Concrete Syntax and Mapping The notation for RequestCause is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at Request, and the static label text ’results in’.

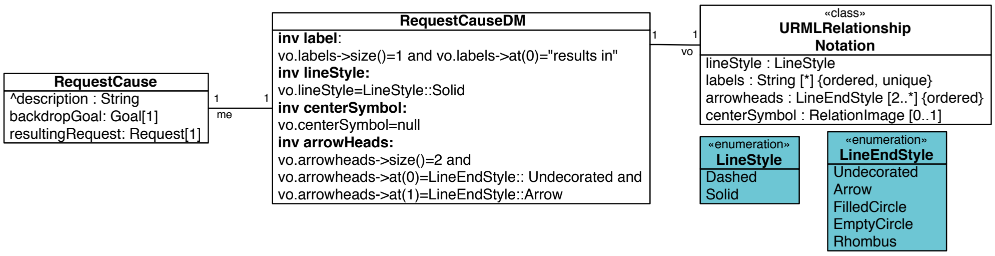

RequestCause
^description : String
backdropGoal: Goal[1]
resultingRequest: Request[1]
1
me
1
RequestCauseDM
inv label:
vo.labels->size()=1 and vo.labels->at(0)="results in"
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle:: Undecorated and
vo.arrowheads->at(1)=LineEndStyle::Arrow
1
vo
1
«class»
URMLRelationship
Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.112.: RequestCause Notation

Images No special imagery.

A.4.69. RequestRealization (Relationship)

Shows which requirement addresses a stakeholder request.

Generalizations

- URMLModelRelationship

Description In order to be able to show stakeholders how their requests are addressed, this relationships enables the mapping of requirements to requests.

Attributes

- No additional attributes.

Role Names and Cardinalities

originatedRequest: Request[1]

resultingRequirement: Requirement[1]

Template Sentence “Feature realizes Request”

Concrete Syntax and Mapping The notation for RequestRealization is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at Requirement, and the static label text ’results in’.

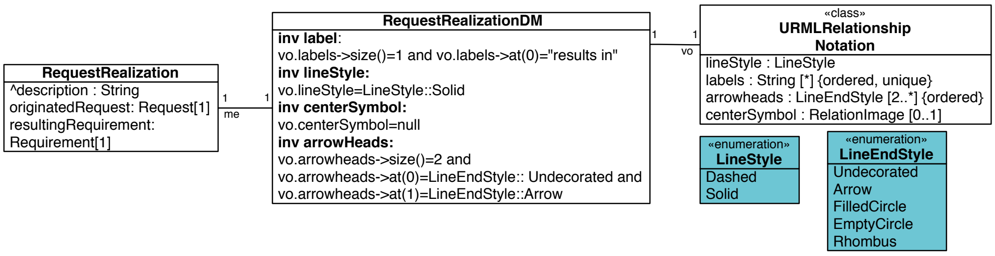

RequestRealization
^description : String
originatedRequest: Request[1]
resultingRequirement:
Requirement[1]
1
me
1
RequestRealizationDM
inv label:
vo.labels->size()=1 and vo.labels->at(0)="results in"
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle:: Undecorated and
vo.arrowheads->at(1)=LineEndStyle::Arrow
1
1
vo
«class»
URMLRelationship
Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.113.: RequestRealization Notation

Images No special imagery.

A.4.70. Requirement (Abstract Class)

Abstract class representing the commonalities of functional and quality requirements.

Generalizations

- URMLModelEntity

Description Requirements are properties or qualities the system needs to fulfill. They can be ranked and prioritized. Requirement itself is an abstract entity, either is specialized to describe desired functionality (i.e. functional requirement) or quality of the system (i.e. quality requirement). Requirements can be involved in the mitigation of dangers (See RequirementMitigation). Requirements can be decomposed into sub-requirements through a refinement relationship. Any requirement can be imposed by a regulatory body. A requirement can be marked being a cost driver.

Attributes

- costDriver: Boolean
- priority: Integer
- rank: Integer
- regulatory: Boolean

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)

A.4.71. RequirementMitigation (Relationship)

Protects a harmed element from danger through a requirement.

Generalizations

- Mitigation

Description When a danger leads to a formulation of a system requirement, this is expressed by a requirement mitigation relationship. This means that the danger is mitigated technically, i.e. a requirement is formulated that makes the system prepared for the danger.

Attributes

- No additional attributes.

Inherited Role Names and Cardinalities

mitigatedDanger (from Mitigation)

protectedHarmedElement (from Mitigation)

Role Names and Cardinalities

mitigatingRequirement: Requirement[1..*]

Template Sentence “Requirement protects HarmedElement from Danger”

Concrete Syntax and Mapping The notation for RequirementMitigation is a relationship notation. It has a solid line, no arrowheads at any line end three different static label texts 'protect', 'from', 'with'. It has the center symbol MitigationImage.

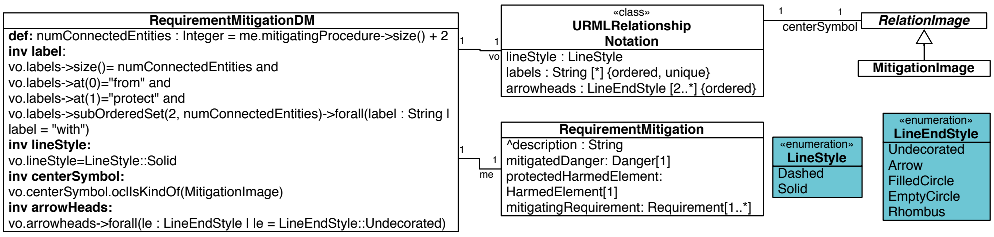

RequirementMitigationDM
def: numConnectedEntities : Integer = me.mitigatingProcedure->size() + 2
inv label:
vo.labels->size()= numConnectedEntities and
vo.labels->at(0)="from" and
vo.labels->at(1)="protect" and
vo.labels->subOrderedSet(2, numConnectedEntities)->forall(label : String I
label = "with")
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol.oclIsKindOf(MitigationImage)
inv arrowHeads:
vo.arrowheads->forall(le : LineEndStyle I le = LineEndStyle::Undecorated)
1
1
vo
1
1
me
«class»
URMLRelationship
Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
1
1
centerSymbol
RelationImage
MitigationImage
RequirementMitigation
^description : String
mitigatedDanger: Danger[1]
protectedHarmedElement:
HarmedElement[1]
mitigatingRequirement: Requirement[1..*]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.114.: RequirementMitigation Notation

Images MitigationImage is presented in Section A.4.47.

A.4.72. RequirementRefinement (Relationship)

Creates a hierarchy of requirements.

Generalizations

- URMLModelRelationship

Description The RequirementRefinement can be used by the modeler to split up coarse-grained requirements into finer ones. This relationship should not be used excessively as the meaning of a large requirement hierarchy might introduce redundancy to model elements expressing similar meaning in a different way, e.g. the FeatureDescriptionRequirement and FeatureConstraint relationships. However, there might be situations in which a coarse-grained requirement is related to a high-level use case, in which the modeler might wish to provide more details about whether a refinement of that requirement is related to an included use case.

Attributes

- No additional attributes.

Role Names and Cardinalities

refiningRequirement: Requirement[1]

refinedRequirement: Requirement[1]

Template Sentence “Requirement refines Requirement”

Concrete Syntax and Mapping The notation for RequirementRefinement is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at refinedRequirement, and the static label text ’refines’.

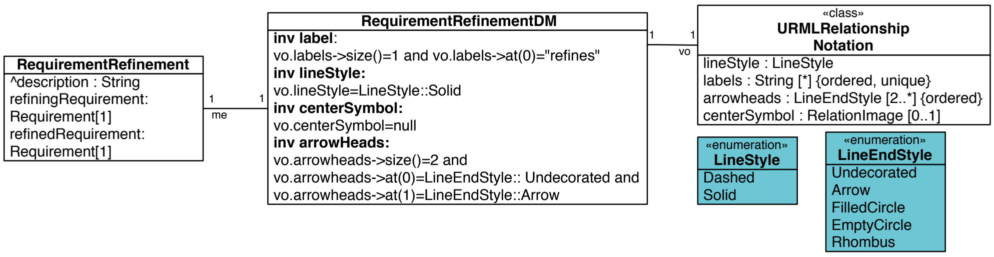


Figure A.115.: RequirementRefinement Notation

Images No special imagery.

A.4.73. ServiceContainment (Relationship)

Indicates which service provider is used by the system under discussion.

Generalizations

- URMLModelRelationship

Description The system under discussion can make use of other systems, for which a black box view is provided through the service provider concept. This relationship connects service providers with the system under discussion. It thus enables modeling the use of components internal to the system. This is important for planning the embedment of commercial-off-the-shelf (COTS) components into the system but also for staffing in systems in which persons play an active role.

Attributes

- No additional attributes.

Role Names and Cardinalities

systemUnderDiscussion: SystemUnderDiscussion[1]

internalService: ServiceProvider[1]

Template Sentence “SystemUnderDiscussion uses ServiceProvider”

Concrete Syntax and Mapping The notation for ServiceContainment is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at ServiceProvider, and the static label text ’uses’.

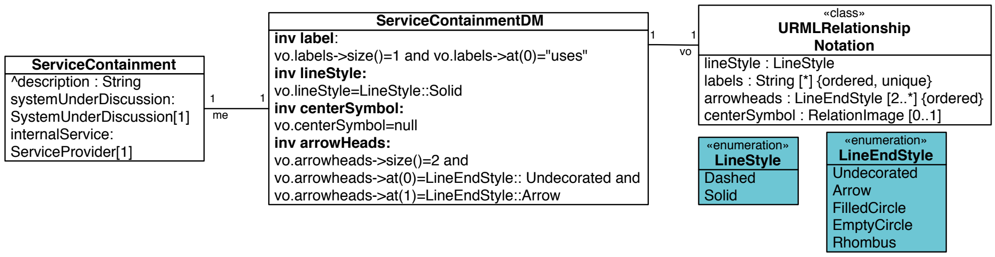


Figure A.116.: ServiceContainment Notation

Images No special imagery.

A.4.74. ServiceProvider (Class)

An internal component of the system that is either another system, software, or a human.

Generalizations

- HarmedElement
- System

Description A service provider is a part of the system under discussion and a system by itself. It offers services to the outside through use cases. Each use case should have at least one controlling service provider instance (a.k.a. 'control object'). As it is part of the system, it may be harmed by a Hazard that affects the system. It is categorized by its ServiceProviderType as either Human, System, or Software.

Attributes

- type: ServiceProviderType

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)

Concrete Syntax and Mapping The notation for ServiceProvider is an entity notation. It uses the name attribute of the ServiceProvider meta-class to label its instances. The description attribute is currently not reflected in the notation. The used image varies depending on the value of the type attribute.

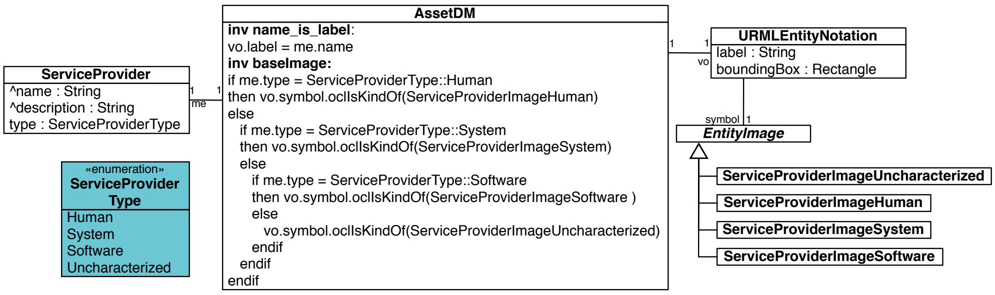

ServiceProvider
^name : String
^description : String
type : ServiceProviderType
1
1
me
«enumeration»
ServiceProvider Type
Human
System
Software
Uncharacterized
AssetDM
inv name_is_label:
vo.label = me.name
inv baseImage:
if me.type = ServiceProviderType::Human
then vo.symbol.oclIsKindOf(ServiceProviderImageHuman)
else
if me.type = ServiceProviderType::System
then vo.symbol.oclIsKindOf(ServiceProviderImageSystem)
else
if me.type = ServiceProviderType::Software
then vo.symbol.oclIsKindOf(ServiceProviderImageSoftware)
else
vo.symbol.oclIsKindOf(ServiceProviderImageUncharacterized)
endif
endif
endif
1
1
vo
URMLEntityNotation
label : String
boundingBox : Rectangle
symbol
1
EntityImage
ServiceProviderImageUncharacterized
ServiceProviderImageHuman
ServiceProviderImageSystem
ServiceProviderImageSoftware

Figure A.117.: ServiceProvider Notation

Images The image, under all circumstances, depicts a platter with two tools on it. If the category is not set, a black box is depicted to the left of the platter1. If the category is 'Human' then a person is carrying the platter. If it is 'System', a robot is carrying the platter. If it is 'Software', the platter stands on top of a computer box, right to a display.

Figure A.118.: ServiceProviderImageUncharacterized

Figure A.119.: ServiceProviderImageHuman

Figure A.120.: ServiceProviderImageSoftware

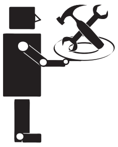


Figure A.121.: ServiceProviderImageSystem

A.4.75. ServiceProviderType (Enumeration)

Categorized Class

- ServiceProvider

Enumeration Literals

- Human
- Software
- System
- Uncategorized

Description

Human - A person is providing the service.

Software - A software system is providing the service.

System - A system is providing the service. The system service provider is used for mixed-type systems, possibly including mechanical components, software components, and humans.

Uncategorized - The type of system is not yet determined.

A.4.76. SoftGoal (Class)

Generalizations

- Goal

Description An objective expressed by a stakeholder that should be achieved by the system in operation. In contrast to a hard goal, a soft goal's realization is not testable.

Attributes

- No additional attributes.

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)

Concrete Syntax and Mapping The notation for SoftGoal is an entity notation. It uses the name attribute of the SoftGoal meta-class to label its instances. The description attribute is currently not reflected in the notation.

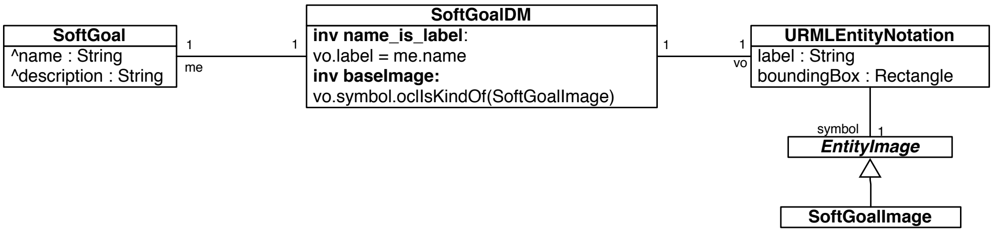


Figure A.122.: SoftGoal Notation

Images The notation uses exactly one image, SoftGoalImage. It depicts a cloud which is a target for an arrow.

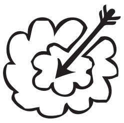


Figure A.123.: SoftGoalImage

A.4.77. Stakeholder (Class)

Generalizations

- URMLModelEntity
- HarmedElement

Description Someone or something (e.g. a regulatory agency) interested in the defined or to be defined system, its product(s) or process(es). A stakeholder can either be an actor, a user of the system, a Customer, who is a person that pays for the system, or a BusinessStakeholder, which is any person with a commercial interest in the project's success. Stakeholders can form hierarchies, which can be expressed by means of the "reports to" relationship. They also have different importance, expressed by the weight attribute. Stakeholder utterances can be formalized in different ways. They can be goals, requests, or ideas. Stakeholders have assets that may be harmed by dangers, which is of importance when evaluating the severity of a danger. The interviewedPersons attribute can be used as a mechanism to track to which concrete persons the requirements analyst talked.

Attributes

- weight: Integer
- interviewedPersons: String [*]

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)

Concrete Syntax and Mapping The notation for Stakeholder is an entity notation. It uses the name attribute of the Stakeholder meta-class to label its instances. The other attributes are currently not reflected in the notation.

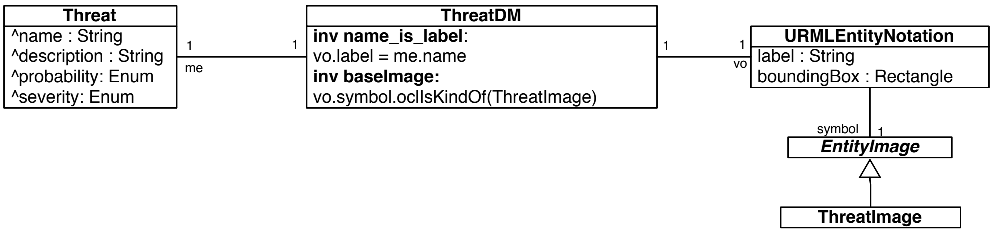


Figure A.124.: Stakeholder Notation

Images The notation uses exactly one image, StakeholderImage. It depicts a person with a speech bubble that has an exclamation mark inside.

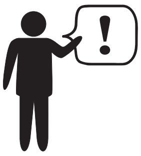


Figure A.125.: StakeholderImage

A.4.78. StakeholderHierarchy (Relationship)

A hierarchical relationship that indicates who reports to whom.

Generalizations

- URMLModelRelationship

Description The stakeholders of a system might themselves have certain relationships to each other. This relationship links two stakeholders, where one stakeholder is reporting to the other. This helps with a trade-off analysis of the different stakeholder's statements.

Attributes

- No additional attributes.

Role Names and Cardinalities

reportingStakeholder: Stakeholder[1]

superiorStakeholder: Stakeholder[1]

Template Sentence “Stakeholder reports to Stakeholder”

Concrete Syntax and Mapping The notation for StakeholderHierarchy is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at superiorStakeholder, and the static label text ’reports to’.

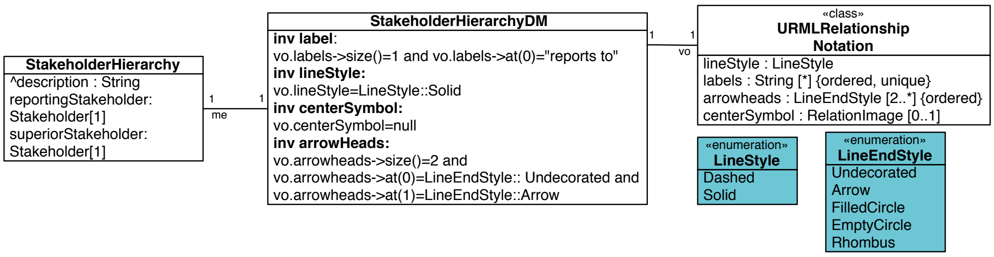


Figure A.126.: StakeholderHierarchy Notation

Images No special imagery.

A.4.79. System (Abstract Class)

Generalizations

- URMLModelEntity

Description A system is an abstract concept that is characterized by its boundary objects (indicating how it can be used). A system can interact with other systems through their boundary objects which enables initiation of or participation in processes of the other system.

Attributes

- No additional attributes.

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)

A.4.80. SystemEmbedment (Relationship)

Shows in which environment process the system under discussion is embedded.

Generalizations

- URMLModelRelationship

Description The system under discussion offers features that enable environment processes. Not all environment processes that are modeled are using the system under discussion. Some of them might be part of the model because they trigger dangers. Thus, there is a need to highlight which of the environment processes actually uses the system.

Attributes

- No additional attributes.

Role Names and Cardinalities

embeddedSystem: SystemUnderDiscussion[1]

environmentProcess: EnvironmentProcess[1]

Template Sentence “SystemUnderDiscussion is used in EnvironmentProcess”

Concrete Syntax and Mapping The notation for SystemEmbedment is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at EnvironmentProcess, and the static label text ’used in’.

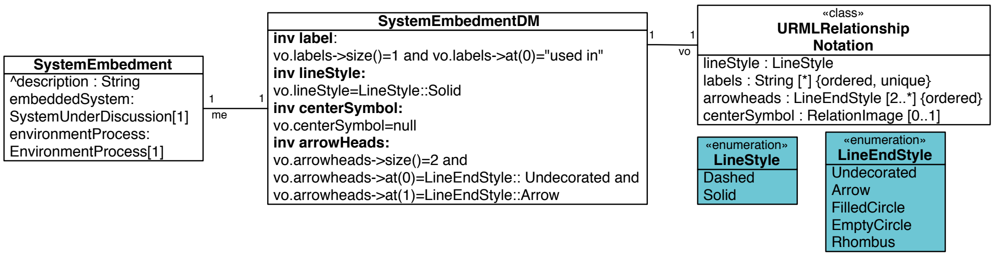


Figure A.127.: SystemEmbedment Notation

Images No special imagery.

A.4.81. SystemFunction (Relationship)

Shows the functions of a system in terms of behavior.

Generalizations

- URMLModelRelationship

Description Apart from the functional requirements, use cases constitute the functional behavior of the system under discussion. This relationship links a use case with the system under discussion it is a function of.

Attributes

- No additional attributes.

Role Names and Cardinalities

systemOfFunction: SystemUnderDiscussion[1]

functionOfSystem: UseCase[1]

Template Sentence “SystemUnderDiscussion has UseCase”

Concrete Syntax and Mapping The notation for SystemFunction is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at UseCase, and the static label text ’has’.

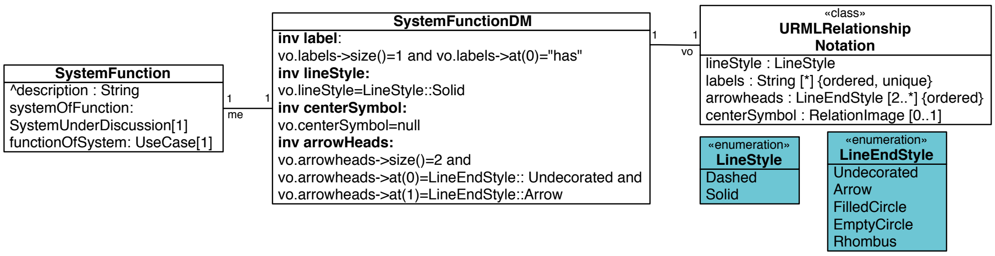

SystemFunction
^description : String
systemOfFunction:
SystemUnderDiscussion[1]
functionOfSystem: UseCase[1]
1
1
me
SystemFunctionDM
inv label:
vo.labels->size()=1 and vo.labels->at(0)="has"
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle:: Undecorated and
vo.arrowheads->at(1)=LineEndStyle::Arrow
1
1
vo
«class»
URMLRelationship
Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.128.: SystemFunction Notation

Images No special imagery.

A.4.82. SystemInteraction (Relationship)

Shows how an external system interacts with the system under discussion’s processes via boundary objects.

Generalizations

- URMLModelRelationship

Description The SystemInteraction relationship describes how systems interact, from the viewpoint of a system under discussion. An instance of such a relationship shows through which boundary object a system initiates a process of another system or takes part in a process of another system. As the internal workings of another system are out of scope for the system under discussion[2], the acting system usually is an instance of a service provider or an actor. The interaction with a service provider shows how the system under discussion uses components of which the modeler only has a black-box view. The interaction with the actor shows how the system under discussion is embedded in its environment.

Attributes

- type: SystemInteractionType

[2] If absolutely needed, these can be modeled in a separate URML model.

Role Names and Cardinalities

actingSystem: System[1..*]

executedProcess: Process[1]

usedBoundaryObject: BoundaryObject[1]

Template Sentence “System interacts with Process via BoundaryObject”

Concrete Syntax and Mapping The notation for SystemInteraction is a relationship notation. It has a solid line, no arrowheads at any line end three different static label texts ’interacts’, ’with’, ’via’. The number of lines labeld with ’interacts’ is determined by the number of connected System instances. The notation uses the center symbol SystemInteractionImage.

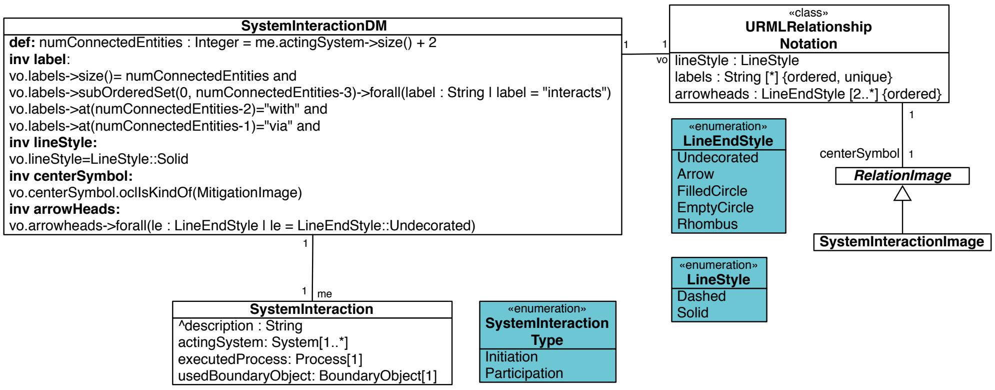

SystemInteractionDM
def: numConnectedEntities : Integer = me.actingSystem->size() + 2
inv label:
vo.labels->size()= numConnectedEntities and
vo.labels->subOrderedSet(0, numConnectedEntities-3)->forall(label : String I label = "interacts")
vo.labels->at(numConnectedEntities-2)="with" and
vo.labels->at(numConnectedEntities-1)="via" and
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol.oclIsKindOf(MitigationImage)
inv arrowHeads:
vo.arrowheads->forall(le : LineEndStyle I le = LineEndStyle::Undecorated)
1
1
vo
«class»
URMLRelationship
Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
1
centerSymbol
1
RelationImage
SystemInteractionImage
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus
«enumeration»
LineStyle
Dashed
Solid
1
1
me
SystemInteraction
^description : String
actingSystem: System[1..*]
executedProcess: Process[1]
usedBoundaryObject: BoundaryObject[1]
«enumeration»
SystemInteraction
Type
Initiation
Participation

Figure A.129.: SystemInteraction Notation

Images SystemInteractionImage depicts three arrows. One has a black fill color and points towards a circle made up of two bended arrows that have white fill color.

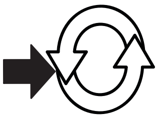


Figure A.130.: SystemInteractionImage

A.4.83. SystemInteractionType (Enumeration)

Categorized Class

- SystemInteraction

Enumeration Literals

- Initiate
- Participate

Description In order to differentiate between process initiation and process participation, the SystemInteraction has an attribute 'type' typed by this enumeration.

A.4.84. SystemInterest (Relationship)

Connects a stakeholder with a system he/she is interested in.

Generalizations

- URMLModelRelationship

Description In stakeholder analysis, a set of stakeholders of the system under discussion is determined. As the URML potentially allows modeling multiple systems at once, there is a need to map stakeholders to systems. This relationship connects a stakeholder with the system under discussion he/she is interested in.

Attributes

- No additional attributes.

Role Names and Cardinalities

interestedStakeholder: Stakeholder[1]

systemOfInterest: SystemUnderDiscussion[1]

Template Sentence "Stakeholder has interest in SystemUnderDiscussion"

Concrete Syntax and Mapping The notation for SystemInterest is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at SystemUnderDiscussion, and the static label text 'has interest in'.

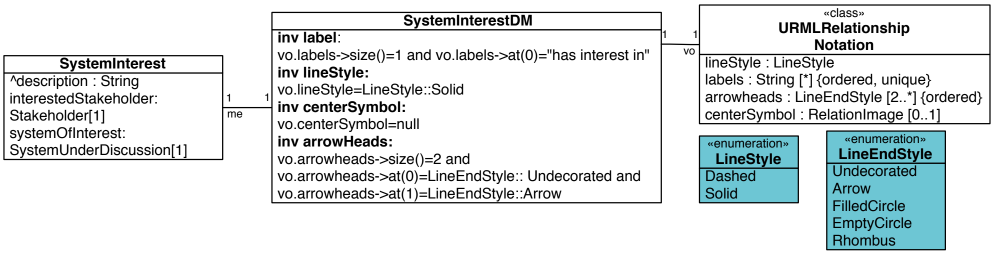


Figure A.131.: SystemInterest Notation

Images No special imagery.

A.4.85. SystemUnderDiscussion (Class)

Generalizations

- System

Description The system under discussion is the central entity of the model. This is the system for which requirements shall be elicited. A system under discussion has stakeholders which have an interest in it. It is embedded in the customer's (which is one of the stakeholders) environment (which is model via EnvironmentProcess). The function it offers to its actors are modeled via use cases. For requirements elicitation, no detailed subsystem decomposition is done yet. It shall however be visible if persons, other systems, or software are relied upon by the system (which is modeled through containment of service providers).

Attributes

- No additional attributes.

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)

Concrete Syntax and Mapping The notation for SystemUnderDiscussion is an entity notation. It uses the name attribute of the SystemUnderDiscussion meta-class to label its instances. The description attribute is currently not reflected in the notation.

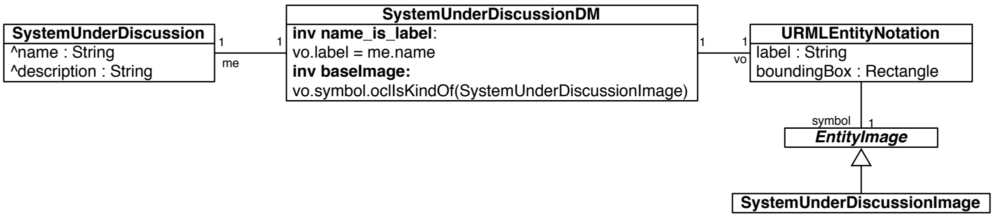


Figure A.132.: SystemUnderDiscussion Notation

Images The notation uses exactly one image, SystemUnderDiscussionImage. It depicts three gears, the ones on the lower left and the upper right only shown partially, both being black. The one in the center is white. The image is bordered by a rectangular bounding box.

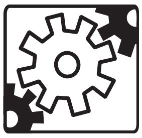


Figure A.133.: SystemUnderDiscussionImage

A.4.86. TestDescription (Relationship)

Connects a goal with a description how it might be tested.

Generalizations

- URMLModelRelationship

Description This relationship connects an assessment sketch with a goal. A goal that has no connected assessment sketch is either incomplete or should be a soft goal.

Attributes

- No additional attributes.

Role Names and Cardinalities

testableGoal: HardGoal[1]

possibleTest: AssessmentSketch[1]

Template Sentence “AssessmentSketch outlines test case of Goal”

Concrete Syntax and Mapping The notation for TestDescription is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at HardGoal, and the static label text ’assesses’.

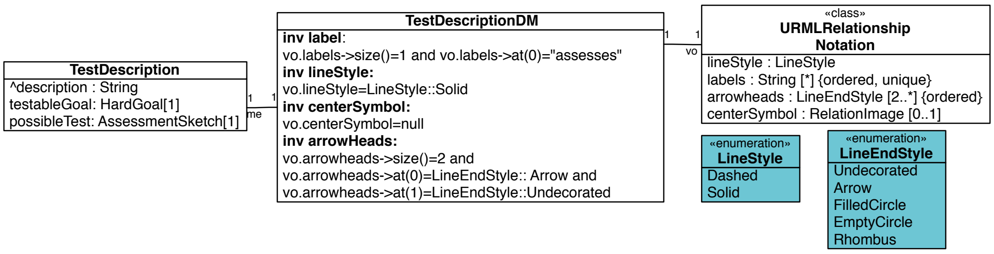

TestDescription
^description : String
testableGoal: HardGoal[1]
possibleTest: AssessmentSketch[1]
1
1
me
TestDescriptionDM
inv label:
vo.labels->size()=1 and vo.labels->at(0)="assesses"
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle:: Arrow and
vo.arrowheads->at(1)=LineEndStyle::Undecorated
1
1
vo
«class»
URMLRelationship Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.134.: TestDescription Notation

Images No special imagery.

A.4.87. Threat (Class)

Generalizations

- Danger

Description Potential cause of loss of assets (e.g. financial as opposed to physical harm). Dangers posing such risk are modeled as threats.

Attributes

- No additional attributes.

Inherited Attributes

- description (from URMLModelElement)
- name (from URMLModelEntity)

Concrete Syntax and Mapping The notation for Threat is an entity notation. It uses the name attribute of the Threat meta-class to label its instances. The other attributes are currently not reflected in the notation.

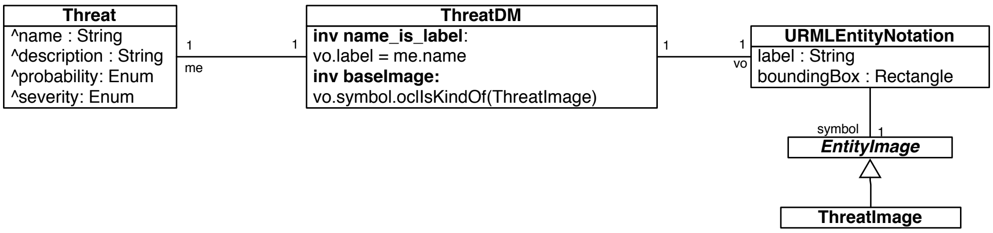


Figure A.135.: Threat Notation

Images The notation uses exactly one image, ThreatImage. It depicts a person's head with a mask and a bag with coins in it.

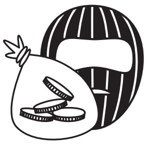


Figure A.136.: ThreatImage

A.4.88. URMLModel (Class)

Generalizations

- None

Description URMLModel is the unique starting symbol used for checking grammatical correctness of an URML model. It also is the top level container of such a model, directly or indirectly containing all its elements.

Attributes

- name: String
- description: String

A.4.89. URMLModelElement (Abstract Class)

Generalizations

- None

Description URMLModelElement is the abstract superclass of all elements of a URML model. Model elements are classified into entities and relationships.

Attributes

- description: String

Inherited Attributes

- None

A.4.90. URMLModelEntity (Abstract Class)

Generalizations

- URMLModelElement

Description URMLModelEntity is the abstract superclass of all classes in the URML meta-model that are not inheriting from another class (e.g. FunctionalRequirement is a subclass of Requirement which in turn is a subclass of URMLModelElement). This allows delineating the URML model entities from other models (e.g. SysML models) in CASE tools that allow the mixing of multiple visual notations.

Attributes

- name: String

Inherited Attributes

- description (from URMLModelElement)

A.4.91. URMLModelRelationship (Abstract Relationship)

Abstract superclass of all relationships defined by the URML.

Generalizations

- None

Description URMLModelRelationship is the abstract superclass of all relationships defined by the URML. In its current version, this means that any URML relationship can be annotated by a textual description. Furthermore, it allows delineating the URML model relationships from other models (e.g. SysML models) in CASE tools that allow the mixing of multiple visual notations.

Attributes

- No additional attributes.

Role Names and Cardinalities

None

A.4.92. UseCase (Class)

Generalizations

- Process
- URMLModelElement

Description A use case describes the use of a system from the perspective of the user(s) (i.e. actors). The actor's interaction with the system (i.e. a use case) always goes through a boundary object. The sequence of steps that a use case entails is modeled through included (always taking place) and extended use cases (optional or exceptional behavior). This is inherited from the abstract superclass Process. Use cases may be part of a business process and thus describe the interactions with other systems. From the business process perspective, a use case provides a white box view into the system. Use cases detail features, which means they describe the functions of the system on a more detailed level.

Attributes

- No additional attributes.

Inherited Attributes

- atomic (from Process)
- businessProcess (from Process)
- description (from URMLModelElement)
- name (from URMLModelEntity)
- preCondition (from Process)
- postCondition (from Process)
- underConstruction (from Process)

Concrete Syntax and Mapping The notation for UseCase is an entity notation. It uses the name attribute of the UseCase meta-class to label its instances. The description attribute is currently not reflected in the notation. Depending on the leaf and underConstruction attributes, overlay images are applied to to base image. If the leaf attribute is true, exactly one instance of LeafOverlayImage is applied to the base UseCaseImage, if false there must be no decoration of that kind. If the underConstruction attribute is true, exactly one instance of UnderConstructionOverlayImage is applied to the base UseCaseImage, if false there must be no decoration of that kind.

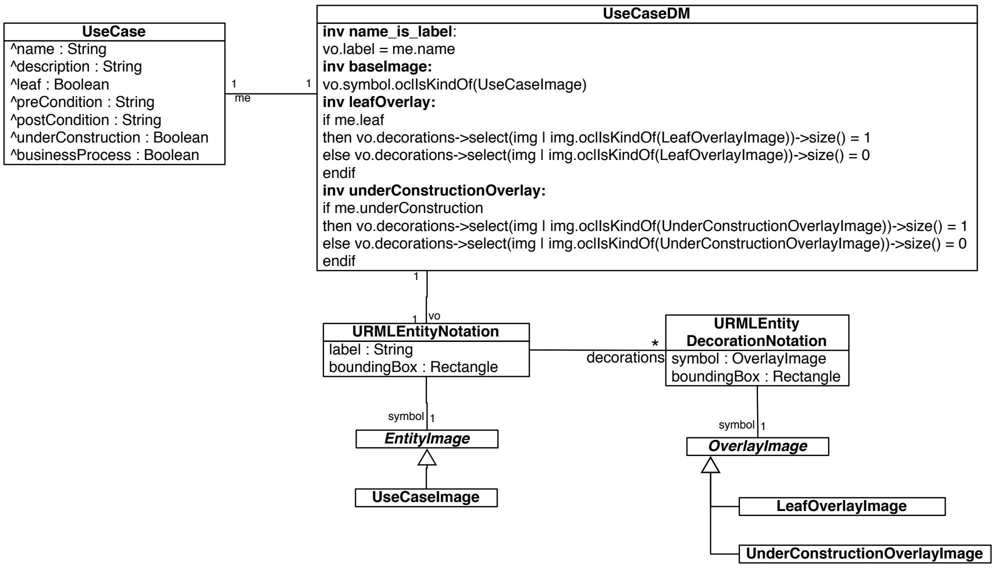

UseCase
^name : String
^description : String
^leaf : Boolean
^preCondition : String
^postCondition : String
^underConstruction : Boolean
^businessProcess : Boolean
1
me
1
UseCaseDM
inv name_is_label:
vo.label = me.name
inv baseImage:
vo.symbol.oclIsKindOf(UseCaseImage)
inv leafOverlay:
if me.leaf
then vo.decorations->select(img | img.oclIsKindOf(LeafOverlayImage))->size() = 1
else vo.decorations->select(img | img.oclIsKindOf(LeafOverlayImage))->size() = 0
endif
inv underConstructionOverlay:
if me.underConstruction
then vo.decorations->select(img | img.oclIsKindOf(UnderConstructionOverlayImage))->size() = 1
else vo.decorations->select(img | img.oclIsKindOf(UnderConstructionOverlayImage))->size() = 0
endif
1
1
vo
URMLEntityNotation
label : String
boundingBox : Rectangle
*
decorations
URMLEntity DecorationNotation
symbol : OverlayImage
boundingBox : Rectangle
symbol
1
EntityImage
UseCaseImage
symbol
1
OverlayImage
LeafOverlayImage
UnderConstructionOverlayImage

Figure A.137.: UseCase Notation

Images The final image depicting UseCase consists of a base image that is decorated by overlays. The base image, UseCaseImage, depicts an oval inspired by the UML use case notation, inside which a person standing in front of a computer is depicted. Below them are two stylized arrows pointing in opposite directions. If the leaf attribute is true, LeafOverlayImage (depicting a filled black circle) is placed on the lower right of the base image. If the underConstruction attribute is true, UnderConstructionOverlayImage (depicting a yellow-and-black striped road barrier) is placed on the lower center.

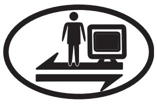


Figure A.138.: UseCase icon

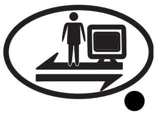


Figure A.139.: Atomic UseCase icon

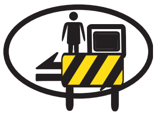


Figure A.140.: Under construction UseCase icon

A.4.93. Vulnerability (Relationship)

Indicates a weak point of a process that might be misused by malicious attacks.

Generalizations

- URMLModelRelationship

Description A vulnerability indicates that a process might be misused and thus could be the source of danger.

Attributes

- No additional attributes.

Role Names and Cardinalities

possibleDanger: Danger[1]

vulnerableProcess: Process[1]

Template Sentence “Process is vulnerable to Danger”

Concrete Syntax and Mapping The notation for Vulnerability is a relationship notation. It has a solid line, no center symbol, an arrow-style arrowhead at the end of line ending at Danger, and the static label text ’vulnerable to’.

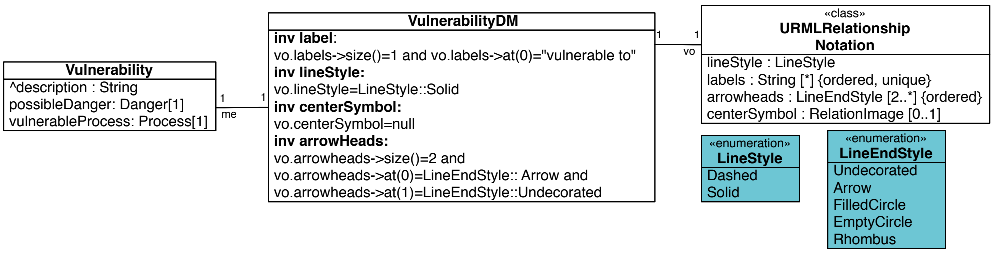

Vulnerability
^description : String
possibleDanger: Danger[1]
vulnerableProcess: Process[1]
1
1
me
VulnerabilityDM
inv label:
vo.labels->size()=1 and vo.labels->at(0)="vulnerable to"
inv lineStyle:
vo.lineStyle=LineStyle::Solid
inv centerSymbol:
vo.centerSymbol=null
inv arrowHeads:
vo.arrowheads->size()=2 and
vo.arrowheads->at(0)=LineEndStyle:: Arrow and
vo.arrowheads->at(1)=LineEndStyle::Undecorated
1
1
vo
«class»
URMLRelationship Notation
lineStyle : LineStyle
labels : String [*] {ordered, unique}
arrowheads : LineEndStyle [2..*] {ordered}
centerSymbol : RelationImage [0..1]
«enumeration»
LineStyle
Dashed
Solid
«enumeration»
LineEndStyle
Undecorated
Arrow
FilledCircle
EmptyCircle
Rhombus

Figure A.141.: Vulnerability Notation

Images No special imagery.

A.5. URML Concrete Syntax Details

This section contains additional figures that complete the concrete syntax model excerpts of Section 4.3. Figures A.142 to A.145 present the concrete subclasses of EntityImage, OverlayImage, RelationImage, and ArrowheadImage.

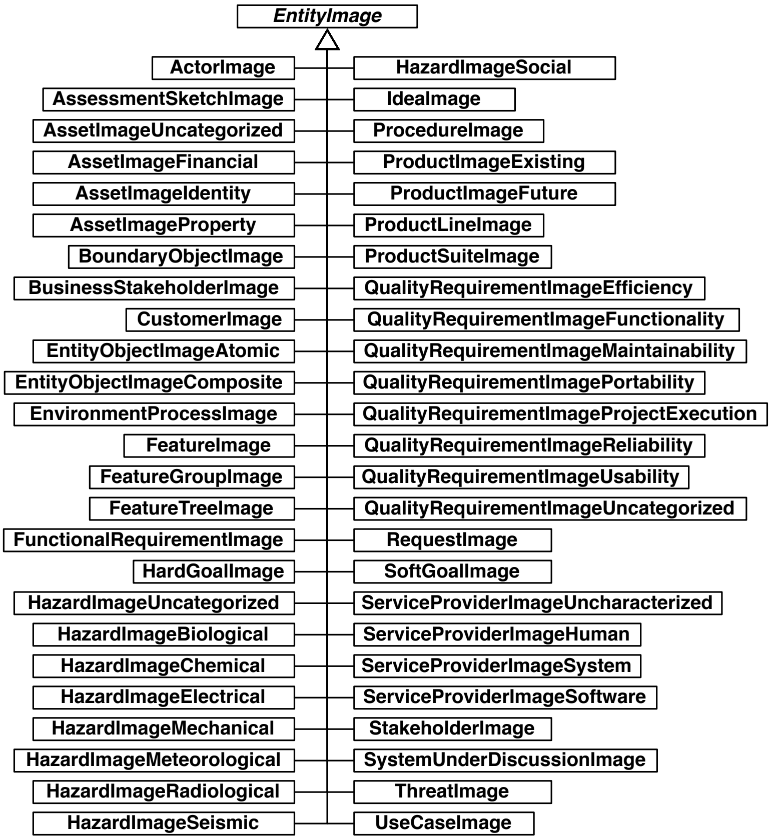


Figure A.142.: Concrete Syntax Model of URML: Basic Structure

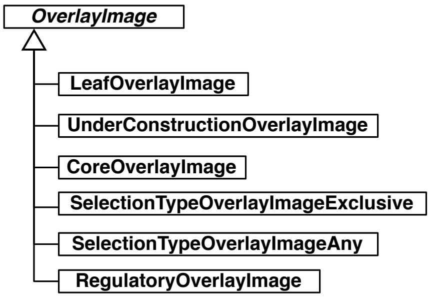

OverlayImage
LeafOverlayImage
UnderConstructionOverlayImage
CoreOverlayImage
SelectionTypeOverlayImageExclusive
SelectionTypeOverlayImageAny
RegulatoryOverlayImage

Figure A.143.: Concrete Syntax Model of URML: Basic Structure

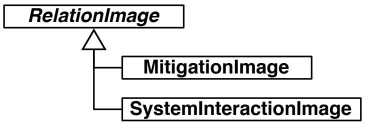


Figure A.144.: Concrete Syntax Model of URML: Basic Structure

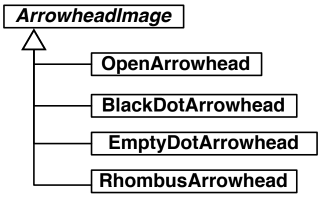


Figure A.145.: Concrete Syntax Model of URML: Basic Structure

A.6. URML Known Issues and Requested Features

Known issues and requested features regarding URML and the reference implementation were collected over the course of this thesis. These do not include issues we have discussed in the main text of this dissertation (e.g. because uncovered during the experiment) and are listed here for the sake of completeness.

A.6.1. Meta-Model

General

- The `underConstruction` attribute potentially is a property of every logical grouping of elements, not only `FeatureGroups` and `Processes`.
- "Out of Scope" attribute is missing
- No abstraction for putting comments into the model

Process and System

- Differentiate between primary and secondary actor (see (Jacobson et al. 1992))
- isEndUser : boolean potential new attribute of Actor to differentiate between Actors that are humans and Actors that are external systems.

- ProcessInclusion should be a subclass of Aggregation or Process should have a constraint that it may not be part of Aggregations
- That UseCases may include EnvironmentProcesses is confusing
- The `SystemInteraction` relationship is confusing.
- Currently an actor can only interact with a process through a boundary. This seems odd in occasions when I do not know yet about the boundary object.
  - Introducing a boundary object type that defaults to "unspecified"
  - Default name for boundary objects derived from actor name and use case name
- InformationFlow might be renamed to EntityFlow or ObjectFlow

Stakeholder and Goal

- GoalDecomposition should be a subclass of Aggregation or Goal should have a constraint that it may not be part of Aggregations
- Business stakeholder is hard to delineate from Customer in the first place, especially because the customer pays and the business stakeholder orders or even influences other financial aspects (e.g. defining the budget of the customer).
- Why is Request only related to Requirement? Can't a request result in any change in the model?
- Why is AssessmentSketch not related to QualityRequirement

Danger

- Actual occurrences (Fault, Incident) of a danger may motivate certain goals, because their impact was so terrible that stakeholders never want them to happen again.
- Are causal relationships between dangers high-level enough to be a topic for this language?
- Procedure is mitigating dangers, but otherwise it is unconnected.

  - Should procedure be somehow connected with actors of the system? It is not a mitigation at all if there is no user role to follow the procedure.
  - Should procedure be connected to environment process. It is actually a plan for parts of the environment process.

- What is the difference between “a threat to an identity asset” and “a threat to a stakeholder”

- multiplicity of triggeredDangers; we have decided that one trigger may be causing a set of dangers, however this makes the notation n-ary, and thus either a symbol is needed or the line junction is undecorated

Feature

- ProductLineAggregation should be a subclass of Aggregation or ProductLine should have a constraint that it may not be part of Aggregations

- `AbstractFeature` should be renamed to FeatureTreeComponent. That would make it much more understandable what its purpose is.

- What does product line containment (one product line containing the other) mean for the feature trees of two product lines?

- Feature could have an optional enumeration to indicate what primarily drove the formulation of that feature. See (Classen, Heymans, and Schobbens 2008, p. 17) for a discussion of what a feature can indirectly represent (marketing-driven, reuse-driven, communication-driven)

- FeatureExclusion vs FeatureRequirement : could be modeled instead as FeatureDependency, with kind {Require, Exclude}, would be extensible for new feature interaction kinds

A.6.2. Notation

- `SystemInteraction` symbol is too complex

- No notation for the `pre-` and `postCondition` of class `Process` exists.

- No notation for `businessProcess` attribute of `Process` exists.

- No notation for `severity` and `probability` of `Danger`

- No notation for `costDriver` of `Requirement`

- No notation for `weight` of `Actor`
- Relationship notation has high degree of symbol overload

A.6.3. Reference Implementation

- Font size of relationship labels should be smaller than the one of entity labels
- QR refines QR broken
- QR refines FR broken
- FeatureGroup requires / excludes FeatureGroup broken
- Goal Decomposition broken
- Displays arcs on feature group without sub features on same diagram
- Shape-Script bugs: "Overweight" actor, Skewed SystemInteraction, SystemUnderDiscussion: sometimes the base icon changes to a more complex one, SystemInterest relationship symbol sometimes with arrow head, sometimes not
- QuickLinker does not support Inheritance between UrmlElements, via toolbar it is possible
- Mitigating Requirements or Procedures could get a dynamic "Mitigation" overlay which is displayed in collapsed state. When exploded via click on that icon, the Mitigation Relationship and connected HarmedElement and Danger will be additionally displayed on the diagram
- Goal contribution line thickness cannot be modified

A.7. URML UML Profile (Reference Implementation)

This section provides the full specification of the URML UML Profile that is described in chapter 6. The stereotypes of the profile are mapped to their corresponding URML meta-class (as listed in section A.4), and the base meta-class of UML they extend. The mapping to URML meta-classes is almost trivial (i.e. one to one) but for ternary relationships, a one-to-one mapping was impossible. As the profile was maintained by students we supervised, some stereotype names slightly deviate from the URML names. For the mapping to UML base meta-classes, we provide a short explanation why this meta-class was chosen. We also explain aspects that are specific to the reference implementation, e.g. additional stereotype attributes needed by Enterprise Architect.

Table A.1.: The Stereotypes of the URML UML Profile

Profile Construct	URML Meta-Class	UML Base Class
Actor	Actor	Actor
AbstractFeature	AbstractFeature	None
AbstractGoal	Goal	None
AggregateFeature	FeatureSelectionOption	Dependency
Assess	TestDescription	Dependency
AssessmentSketch	Assessment	Class
Asset	Asset	Class
BoundaryObject	BoundaryObject	Class
BoundaryObjectContainment	BoundaryObjectContainment	Association
BusinessStakeholder	BusinessStakeholder	Actor
Constrain	ProcessConstraint and FeatureConstraint	Dependency
Contribute	GoalContribution	Dependency
Customer	Customer	Actor
DerivationTarget	-	None
DerivationSource	-	None
Danger	Danger	None
Derive	-	Dependency
Detail	FeatureDescriptionRequirement and FeatureDescriptionUseCase	Dependency
DetailSource	-	None
DetailTarget	-	None
Enable	ProcessEnabling	Dependency
EntityObject	EntityObject	Class
EnvironmentProcess	EnvironmentProcess	UseCase
Exclude	FeatureExclusion	Dependency
Express	GoalStatement and RequestStatement	Dependency
Feature	Feature	Component
FeatureGroup	FeatureGroup	Component

Table A.1.: The Stereotypes of the URML UML Profile

Profile Construct	URML Meta-Class	UML Base Class
FunctionalRequirement	FunctionalRequirement	Class
Goal	HardGoal	Class
Harm	Harm	Dependency
HarmedElement	HarmedElement	None
Hazard	Hazard	Class
Idea	Idea	Class
Interact	InformationFlow	InformationFlow
Mention	Mention	Dependency
Mitigate	-	Association
Mitigation	Mitigation	AssociationElement
MitigatingProcedure	MitigatingProcedure	Class
Motivate	Motivation	Dependency
Own	AssetOwnership	Dependency
Precede	ProcessPrecedence	Dependency
Presuppose	FeatureRequirement	Dependency
Process	Process	None
Product	Product	Component
ProductLine	ProductLine	Component
ProductSuite	ProductSuite	Class
Protect	-	Dependency
QualityRequirement	QualityRequirement	Class
Report	StakeholderHierarchy	InformationFlow
Realize	GoalRealization	Dependency
RealizationSource	-	None
RealizationTarget	-	None
Refine	RequirementRefinement	Dependency
Represents	FeatureTree	Dependency
Request	Request	Class
Require	ProcessRequirement	Dependency
Requirement	Requirement	None
RequirementSource	-	None
RequirementTarget	-	None
ServiceContainment	ServiceContainment	Dependency
ServiceProvider	ServiceProvider	Class
SoftGoal	SoftGoal	Class

Table A.1.: The Stereotypes of the URML UML Profile

Profile Construct	URML Meta-Class	UML Base Class
Stakeholder	Stakeholder	Actor
System	System	None
SystemEmbedment	SystemEmbedment	Association
SystemFunction	SystemFunction	Association
SystemInteract	-	Association
SystemInteraction	SystemInteraction	AssociationElement
SystemInterest	SystemInterest	Association
SystemUnderDiscussion	SystemUnderDiscusssion	Class
Threat	Threat	Class
Trigger	DangerTrigger	Dependency
UrmlElement	UrmlModelEntity	None
UrmlRelationship	UrmlModelRelationship	None
UseCase	UseCase	UseCase
Vulnerable	Vulnerability	Dependency

A.8. UML Classes as a Basis for Extension

A.8.1. Relationship Base Classes

The relationship-related classes of UML are rooted in the abstract class `Relationship` (Fig. A.146). Beyond rooting the hierarchy, `Relationship` does not have specific semantics or notation, and it does only define one association to `Element` called `relatedElements`, which is a derived union standing for the `Elements` connected by the relationship. `Relationship` is specialized by `DirectedRelationship` and `Association`.

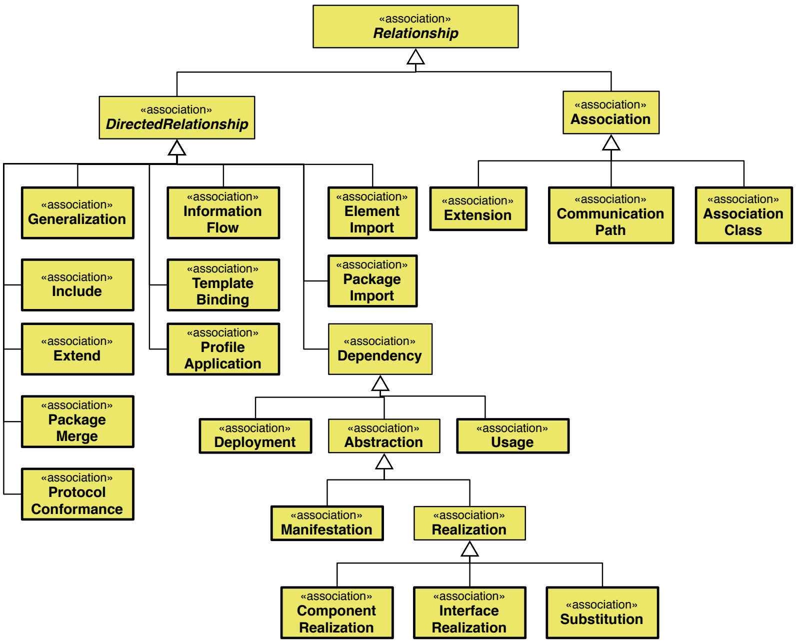

«association» Relationship
«association» DirectedRelationship
«association» Association
«association» Generalization
«association» Information Flow
«association» Element Import
«association» Extension
«association» Communication Path
«association» Association Class
«association» Include
«association» Template Binding
«association» Package Import
«association» Extend
«association» Profile Application
«association» Dependency
«association» Package Merge
«association» Deployment
«association» Abstraction
«association» Usage
«association» Protocol Conformance
«association» Manifestation
«association» Realization
«association» Component Realization
«association» Interface Realization
«association» Substitution

Figure A.146.: All UML relationship meta-classes.

DirectedRelationship partitions relatedElements into source and target Elements to represent the direction of the relationship. Apart from this it is as abstract as Relationship is. More concrete specification what source and target can be is provided by concrete subclasses of DirectedRelationship. An example of such a concretization in a subclass are the properties general and specific of Generalization, as shown in Figure A.147. Here, general subsets target and restricts its multiplicity to exactly one, specific does the same for source. Both also refine the type of source and target to Classifier. More technical details regarding how the other subclasses of relationship concretize source and target can be found in (OMG 2011c).

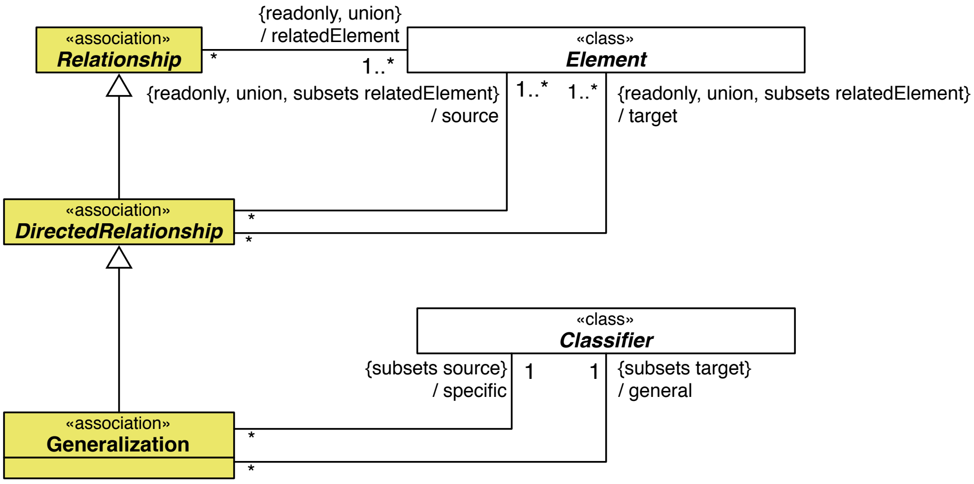


Figure A.147.: All UML relationship meta-classes. Classes that are leaves of the inheritance hierarchy are displayed with a bold border.

`Generalization` is a binary directed relationship between two `Classifiers`, where the specific classifier inherits features and constraints from the general classifier. `Include` and `Extend` both are binary directed relationships between `UseCases`. `Include` inserts the behavior of the included use case to the behavior of the including use case. It means that the included behavior is always executed, i.e. not optional. `Extend` does the same with a different meaning, the behavior is optional. `ElementImport` is a binary directed relationship between a `Namespace` and a `PackageableElement`, which makes the imported element known (i.e. referable by name) to the importing namespace. `PackageImport` is a binary directed relationship between a `Namespace` and a `Package`, which makes the contents of the imported package known to the importing namespace, which is equivalent to a set of `ElementImport` relationships. `PackageMerge` is a binary directed relationship between two packages, where the receiving package's elements are merged with the elements of the package-to-be-merged. `ProfileApplication` is a binary directed relationship between a package and a `Profile`, where the profile's stereotypes extend the corresponding base classes of contained in the extended package. `ProtocolConformance` is a binary directed relationship between `ProtocolStateMachines`, with the specific state machine conforming to the protocol of the the general state machine. `InformationFlow` is an n-ary directed relationship between `NamedElements` where the information sources provide information to the information targets. `TemplateBinding` is a binary directed relationship between `TemplateableElement` and `TemplateSignature`, that binds a certain signature to an element. Dependency is an n-ary directed relationship between NamedElements, where the client elements depend on the supplier elements. The nature of the dependency is kept vague, it may be a semantic or a structural dependency. Its specializations are Usage, Deployment, and Abstraction. Usage models implementation dependencies to show that one element can not operate without the other. Deployment maps artifacts to deployment targets, either on the "type level" or the "instance level" (OMG 2011c, p207). Deployments are used to provide a hardware-software mapping as described in (Bruegge and Dutoit 2009) , but can also map software to software, e.g. to visualize which artifacts constitute a certain software component. Abstraction relationships related elements that represent the same concept seen on different levels of abstraction. It is further specialized to Manifestation, which allows to map parts of a model to a model of the file that would manifest the parts, and different kinds of Realization. Realizations model the relationship between a specification and its implementation. InterfaceRealization shows what model element implements an interface, ComponentRealization shows what model element implements a component, and Substitution allows to model runtime substitutability, i.e. if a classifier can stand in for another classifier at runtime because it adheres to the contract of the substituted classifier.

As opposed to DirectedRelationships, Associations can only relate typed elements that have properties. They enable fine grained expression of the navigability of association ends, and whether the properties are owned by the association or the related element. As Association also inherits from Classifier (not shown on above diagram), associations support inheritance. A very special kind of Association, Extension, has been explained in the previous section. CommunicationPath is used "to model the exchange of signals and messages" (OMG 2011c, p206) between deployment targets. AssociationClass combines the semantics of Class and Association. It adds the ability Class to relate other classes, or, vice versa, the ability to Association to own other properties than the member ends it owns.

A.8.2. Entity Base Classes

It is impossible to provide a short overview on UML entity base classes. UML 2.4.1 contains 242 meta-classes. Subtracting the 25 meta-classes that are used to model relationships, 217 classes remain to be discussed for this section. We do not intend to repeat published literature on UML that already provides an overview as e.g. Fowler (Fowler 2004). This section shall provide terminology to the technical discussion of our URML-to-UML-mapping in section 6.3.

A minor but not unimportant side-note regarding our differentiation of meta-classes into entity and relationship meta-classes: In UML, the distinction between relationship or entity semantics is not always possible. E.g. the root class of all UML meta-classes, `Element`, is so abstract that both `Relationship` and `NamedElement`, the root of most entity meta-classes inherit from it (Fig. A.148). But also lower level constructs like `Classifier` defy a strict categorization: It is both the superclass of `Class` and of `Association` (Fig. A.161).

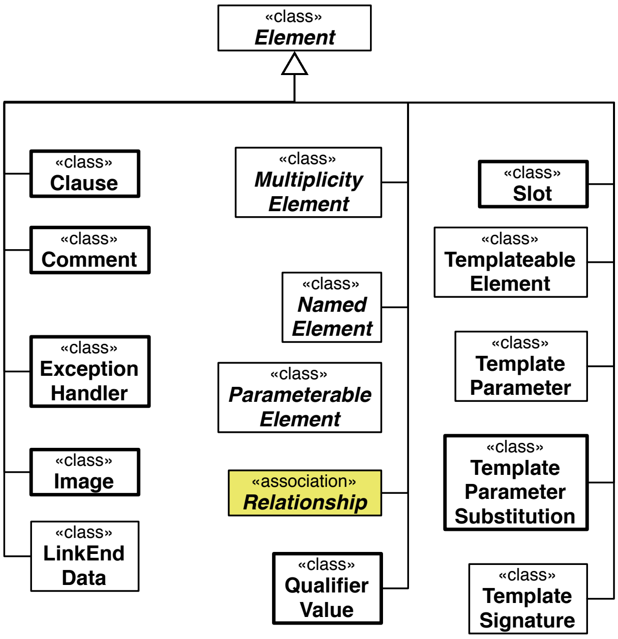


Figure A.148.: The top-level classes of UML.

As mentioned, the root class of UML is `Element`. It constitutes the common core of anything that can be put in a model. It can own other elements or be owned by other elements. In addition, it can own comments. `Comment`, one of its subclasses, adds information to the model that might be useful to the reader but has no defined semantics. Element has 14 more subclasses. Most these 14 have no more subclasses or at most one level of subclasses below (See Fig. A.149).

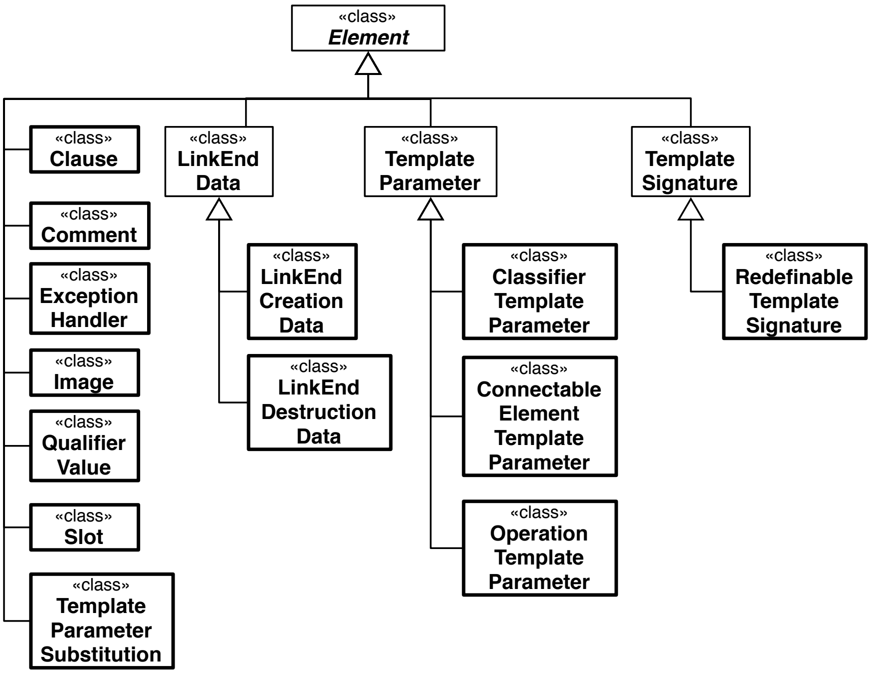


Figure A.149.: Subclasses of Element with no subclasses or not more than one additional level.

Apart from `Comment`, which is used very often on diagrams, we do not discuss the other subclasses in great detail as they are specialized parts of UML packages best explained by more abstract concepts. `Image` provides the serialized presentation of images used for stereotype icons. `Clause`, `ExceptionHandler`, `LinkEndData` including subclasses, and `QualifierValue` are concepts in the scope of activity diagrams. `TemplateParameter` and subclasses, `TemplateParameterSubstitution`, `TemplateSignature`, and `RedefinableTemplateSignature` are part of the Templates package. We will shed a little light on these concepts when discussing `TemplateableElement` and `ParameterableElement`. The other subclasses of `Element` are the roots of deeper inheritance hierarchies (Fig. A.150). `Relationship` already was discussed in the previous section. The other four introduce the concepts of templateability, multiplicity, and namedness.

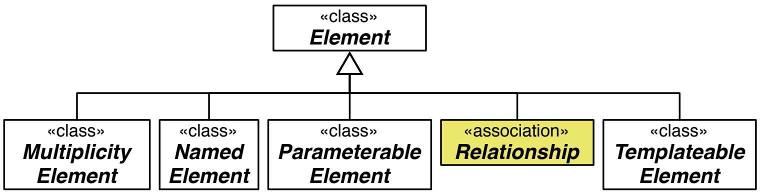


Figure A.150.: Subclasses of Element with no subclasses or not more than one additional level.

TemplateableElement represents classes that can have TemplateParameters, e.g. generic types in Java or template classes in C++. As we can can see in Figure A.151, Classifiers, Operations, Packages, and StringExpressions can be templates. A Classifier "describes a set of instances that have features in common" (OMG 2011c, p.67) (see also Figs. A.160 and A.161). An Operation is a behavior-encapsulating feature of a classifier. A Package is a logical grouping of elements that can be packaged (PackageableElement). PackageableElement will be explained in one of the next paragraphs (see also Fig. A.160). StringExpression is a specialized Expression that can be used for the naming of named elements, potentially derived from TemplateParameters.

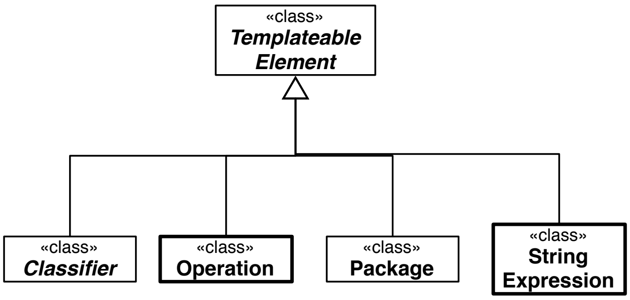


Figure A.151.: TemplateableElements.

ParameterableElement stands for entities that can be "can be exposed as a formal template parameter" (p.629). PackageableElements, ConnectableElements, and Operations are parameterable and thus can be template parameters (Fig. A.152)[3] of TemplateableElements.

[3] As Figure A.149 indicates, PackageableElements are only in theory parameters, the only concrete subclasses of TemplateParameter work for Classifiers (which is a PackageableElement, see Figure A.160), ConnectableElements and Operations.

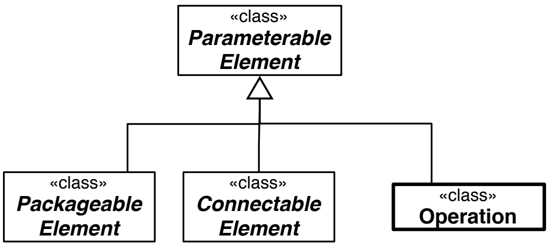


Figure A.152.: ParameterableElements.

MultiplicityElement represents the abstract notion of everything that needs to define multiplicity, e.g. Properties of Classes. Property as a MultiplicityElement provides the bounds for how many instances of the property's Class (a Property is typed by a Class) may be instantiated when instantiating the Class that owns the Property. ConnectorEnds work similarly in composite or internal structure diagrams and may be related to a Property. Pins and Variables show multiplicity of Activities, and Parameters in Operations.

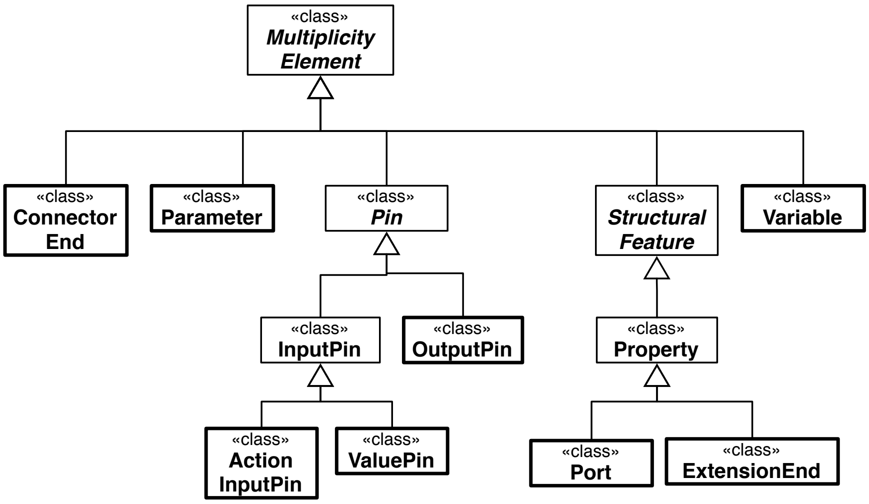


Figure A.153.: MultiplicityElements.

We continue inspecting the subclasses of NamedElement (Fig. A.154). It constitutes the root meta-class of anything that can be named in a UML model. As we did with Element, we divide its subclasses into those that root deeper inheritance hierarchies and those that have no subclasses or only few levels below. Of its 18 subclasses, Namespace, PackageableElement, RedefinableElement, and TypedElement root inheritance hierarchies of more than two levels. The other 14 have fewer or zero levels of inheriting classes below. This time, as opposed to Figure A.149, we omit showing these inheriting classes to avoid cluttering of the diagram, and will proceed so for the remainder of this section without further notice. We also stop discussing the classes that have less than two subclasses. We think the basic principles of UML can be learned from looking at the (mostly abstract) meta-classes that are a superclass to many others.

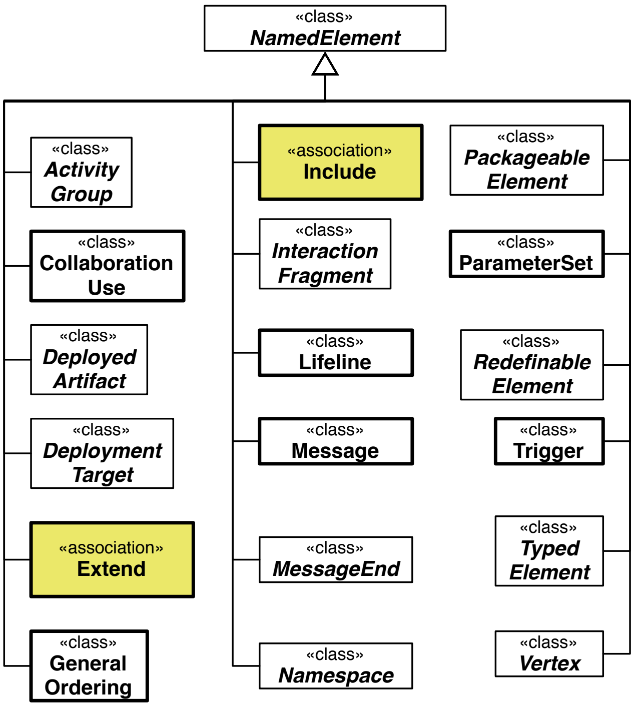


Figure A.154.: NamedElements.

Namespaces enclose PackageableElements and provide a name prefix to those. Namespaces allow to have multiple model elements with the same name in a model (as long as they are in a different namespace). They are a mechanism to deal with complexity. A well-known subclass of Namespace is Package, which is often used to logically group elements in a model that have great coupling. Because a namespace in effect extends the name of its enclosed elements, itself needs to be named. RedefinableElements are elements that when defined in a Classifier, can be redefined in a specializing classifier. TypedElements represent elements that have a Type, i.e. the Type limits the range of valid values the element can have.

By now we can see that the UML inheritance hierarchy is not a tree. To many meta-classes, more than one paths lead from the top-level Element. E.g. `PackageableElement` is not only a subclass of `NamedElement` (Fig. A.155) but also of `ParameterableElement` (Fig. A.152).

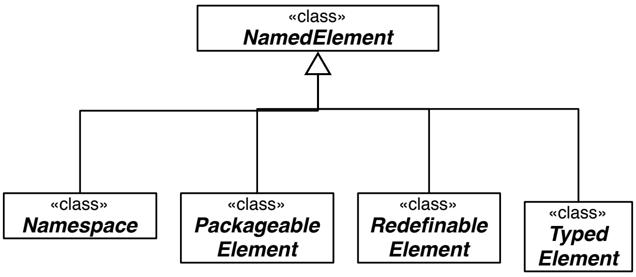


Figure A.155.: NamedElements with deep inheritance hierarchies

UML's inheritance hierarchy is much deeper than this. We have to paint some more diagrams until arriving at diagrams where no class has more than two additional levels of subclasses. To arrive e.g. at Figure A.163, we continue by looking at `RedefinableElement` (Fig. A.158), `Namespace` (Fig. A.159), or `PackageableElement` (Fig. A.160) subclasses, via `Classifier` (Fig. A.161) and its subtypes (Fig A.162). Traversing the hierarchy, we encounter mostly concepts that programmers are familiar with: properties, operations, parameters, states, transitions, packages, classes, events, data types, interfaces, deployed artifacts, components, devices, or state machines.[4] This is not surprising, as UML is a language dedicated to the modeling of software systems. But we already see the difficulties that may arise when using UML as a basis for a domain specific language that is not alone focused on software design and implementation. Most parts of UML can be used to model existing or future properties of software systems. Either they are a description of reality, or can be transformed into source code to become reality. As we are concerned with the modeling of requirements in this thesis, we go on (after Fig. A.164) inspecting meta-classes of UML that are more focused on intended properties of a system. The difference between future and intended properties is that intended properties should be mapped to other UML constructs before source code can be derived from the model.

[4] The list is neither exhaustive nor in any particular order.

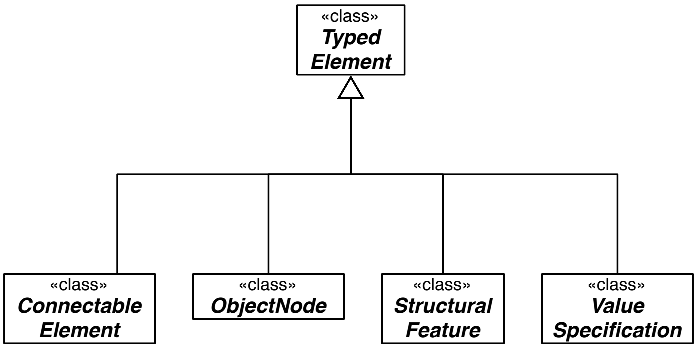


Figure A.156.: TypedElements.

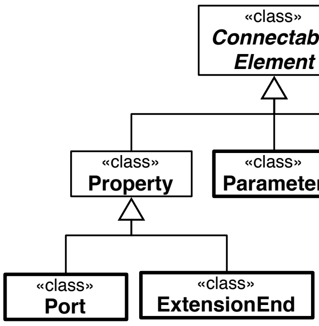


Figure A.157.: ConnectableElements.

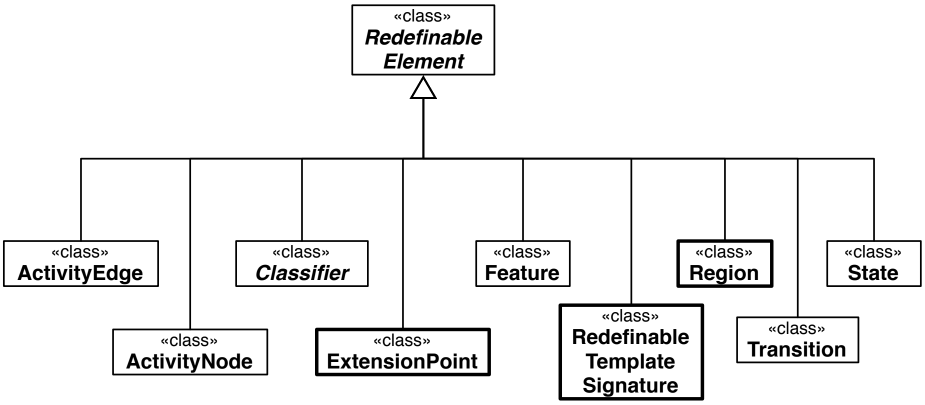


Figure A.158.: RedefinableElements.

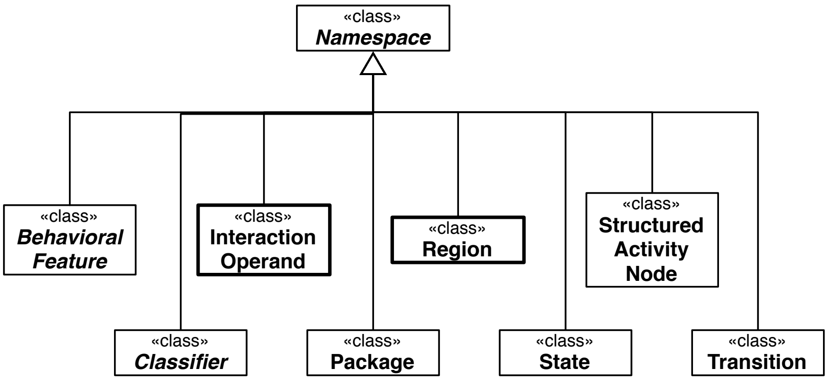


Figure A.159.: Namespaces.

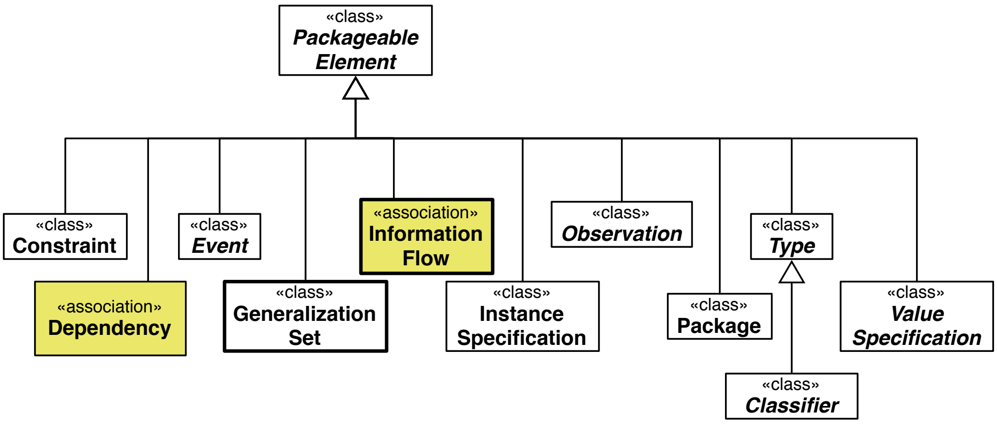

«class» Packageable Element
«class» Constraint
«class» Event
«association» Information Flow
«class» Observation
«class» Type
«association» Dependency
«class» Generalization Set
«class» Instance Specification
«class» Package
«class» Value Specification
«class» Classifier

Figure A.160.: PackageableElements.

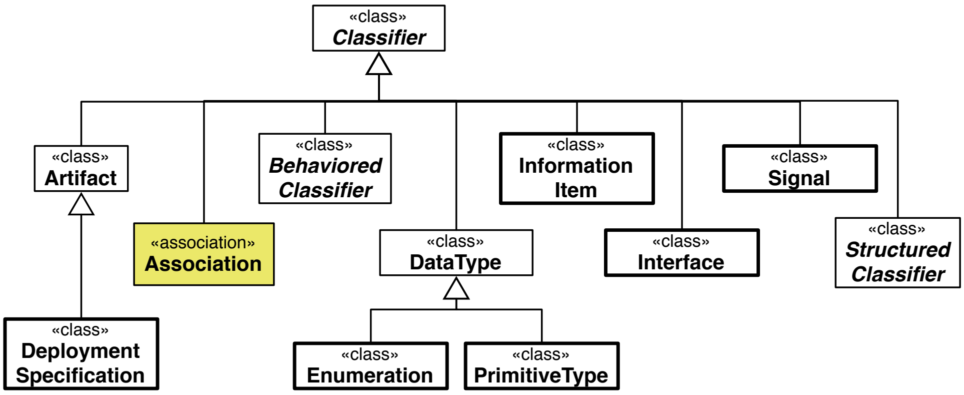


Figure A.161.: Classifiers.

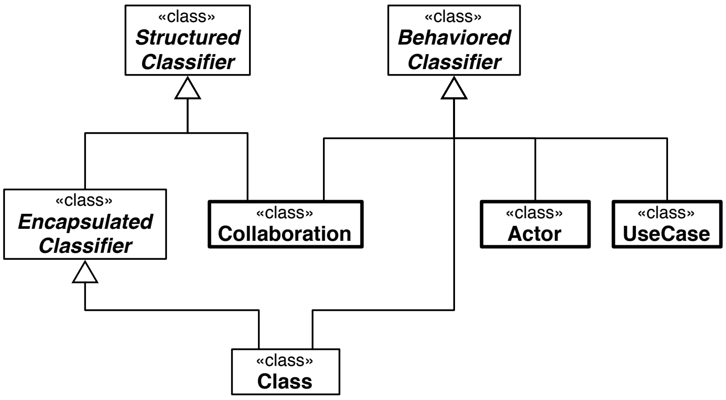


Figure A.162.: Structured- and BehavioredClassifiers.

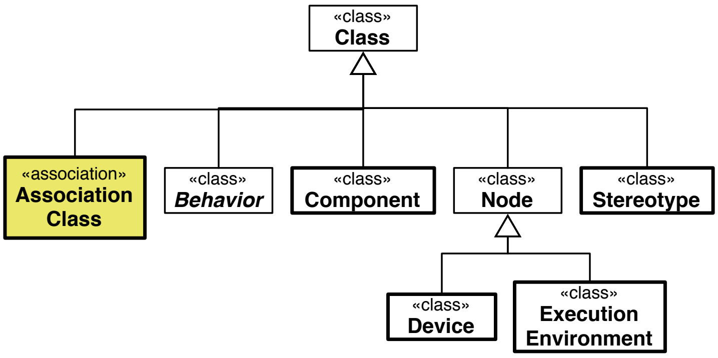


Figure A.163.: Classes.

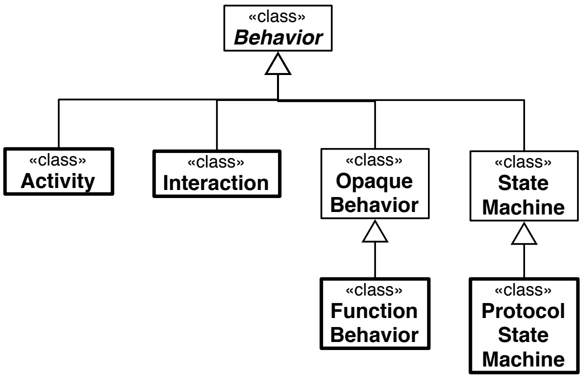


Figure A.164.: Behaviors.

The major UML meta-class that serves to modeling intended properties is the `UseCase`. It is the UML construct to relate a software system to its functional requirements. It describes on a high-level some functionality that the system shall have. `UseCases` can be decomposed via `Include` relationships. Optional or exceptional functionality can be modeled via `Extend` relationships. Before such models can be mapped to source code, use cases have to be mapped to other model elements, describing the structure and behavior of the software system. For example, Bruegge and Dutoit(Bruegge and Dutoit 2009) provide heuristics to derive entity, boundary, and control objects from use cases. Such mappings are supported by UML by the `subject` property of `UseCase`: Any instance of `Classifier` can be the `subject` of a `UseCase` (see Fig. A.165).

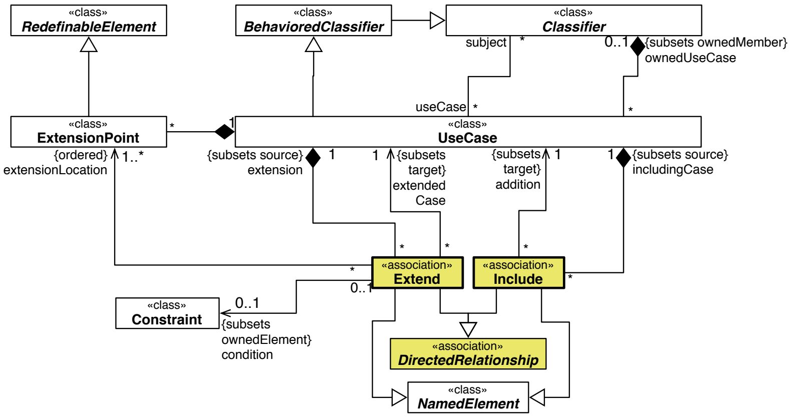
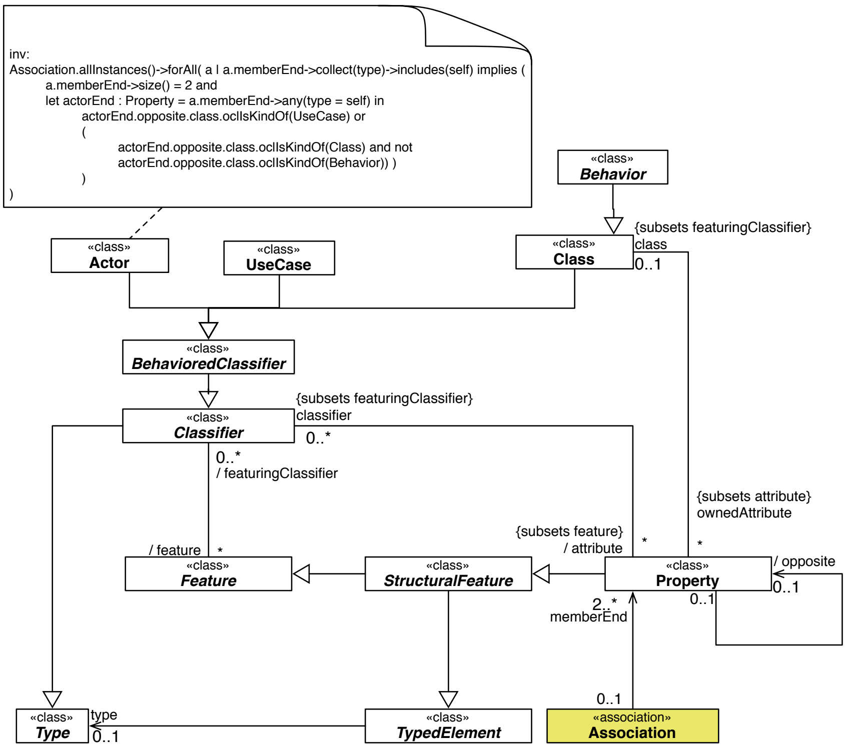


Figure A.165.: Excerpt of UML meta-model with focus on UseCase.

A role of somebody or something external to the system under discussion, that is interacting with the system is modeled by the Actor concept in UML. Conceptually, the usage of the system under discussion is highlighted by an association between Actor and UseCase. UML has no separate relationship type for that purpose, but uses `Association` and places a constraint on the associations in which Actor instances participate. In our opinion this shows the weakness of UML's deep inheritance hierarchy, that is optimized for software design: By virtue of being a BehavioralClassifier, an Actor instance could be related to many things. But as the UML allows Actor instances only in binary associations with instances of UseCase and only certain subtypes of Class, a complicated constraint has to ensure that no invalid connections are created in tools.[5]



```
inv:
Association.allInstances()->forAll( a | a.memberEnd->collect(type)->includes(self) implies (
        a.memberEnd->size() = 2 and
        let actorEnd : Property = a.memberEnd->any(type = self) in
                actorEnd.opposite.class.oclIsKindOf(UseCase) or
                (
                        actorEnd.opposite.class.oclIsKindOf(Class) and not
                        actorEnd.opposite.class.oclIsKindOf(Behavior)) )
                )
)
```

Figure A.166.: Excerpt of UML meta-model with focus on Actor. The goal is to understand the constraint.

[5] Fig. A.166 will use the constraint as specified in (OMG 2013b) as that constraint has one bug less than the one in the specification of (OMG 2011c).

A.9. KAOS Relationship Notation

Table A.2.: Analysis of KAOS relationship notation

Abstraction Name	Line Style	Adornments	Arrowheads	Uniqueness
Refinement	Solid	Circle Adornment, black fill for complete Refinements	Filled black arrowhead	Unique
O-Refinement	Solid	Circle Adornment, black fill for complete Refinements	Filled black arrowhead	Same
Divergence-GoalConflict	Solid	Lightning Symbol Adornment	-	Unique
Divergence-Obstruction	Solid	Orthogonal Line Adornment	Filled black arrowhead	Unique
Resolution	Solid	Orthogonal Line Adornment	Filled black arrowhead	Same
Association	Solid	- , or Diamond Adornment for n-ary assocations	-	Unique
Association-Built-In-Specialization	Solid	-	Triangle arrowhead	Unique
Association-Built-In-Aggregation	Solid	-	Diamond arrowhead	Unique
Dependency	Solid	D-Shaped Adornment	-	Unique
Assignment	?	?	?	?
Monitoring	Solid	-	Open Arrowhead	Unique
Control	Solid	-	Open Arrowhead	Same
Responsibility	Solid	Non-fill circle	-	Unique
Operationalization	Solid	Non-fill-circle	Filled black arrowhead	Same
Input	Solid	-	Open Arrowhead	Same

Table A.2.: Analysis of KAOS relationship notation

Abstraction Name	Line Style	Adornments	Arrowheads	Uniqueness
Output	Solid	-	Open Arrowhead	Same
Performance	Solid	Non-fill circle	-	Same
Instance	?	?	?	
Class Coverage	?	?	?	
Coverage	?	?	?	
Instance Coverage	?	?	?	
BehaviorOf	?	?	?	
History	?	?	?	
ControlledVariable	?	?	?	
Path	?	?	?	
Sequential Composition	?	?	?	
Parallel Composition	?	?	?	
Input(Transition)	Solid	-	Filled black arrowhead	Unique
Output(Transition)	Solid	-	Filled black arrowhead	Same
Label	?	?	?	
Episode	?	?	?	
Model Annotation to Model Element relationship	Dashed	-	-	Unique

A.10. Experiment Appendix

A.10.1. Symbols used in Experiment

In the following sections, all symbols that were tested in the experiment comparing the notations of URML, KAOS, and URN are presented.

A.10.1.1. URML

In the following figure, the symbols of URML's notation together with the term derived from meta-class name and attributes are shown. The symbols are sorted alphabetically by the terms. Not all possible combinations of `Quality Requirement` and `Process` attributes, for example a "regulatory performance quality requirement" were included.

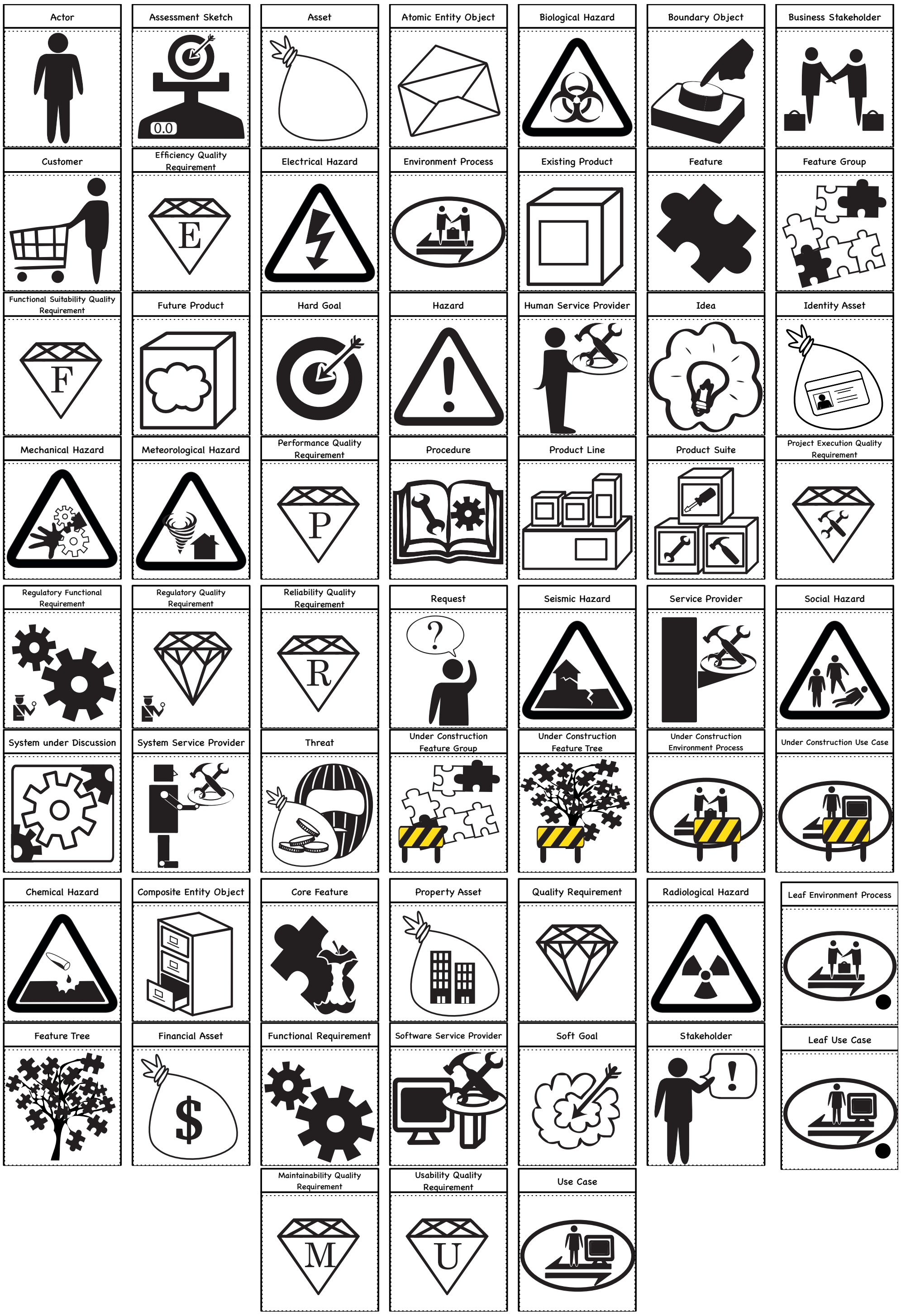


Figure A.167.: Notation tested in URML experiment

A.10.1.2. KAOS

In the following figure, the symbols of KAOS' notation together with the term derived from meta-class name are shown. The symbols are sorted alphabetically by the terms.

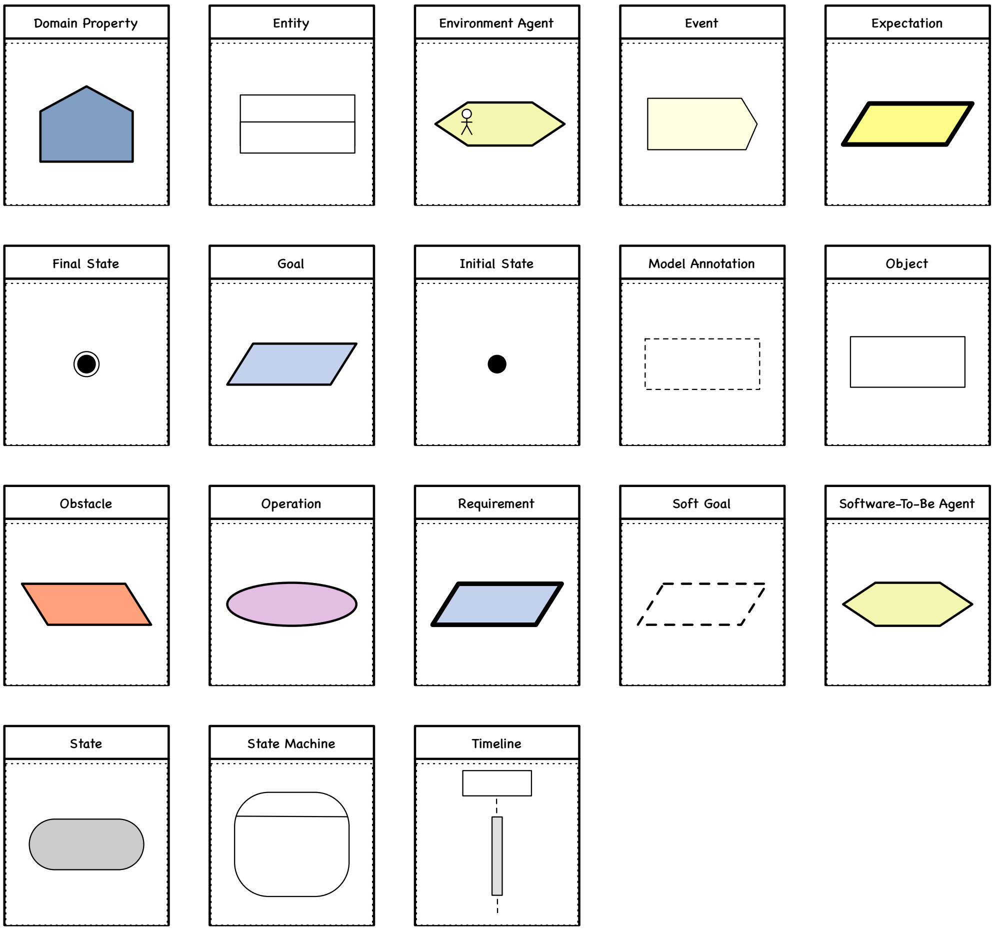


Figure A.168.: Notation tested in KAOS experiment

A.10.1.3. URN

In the following figure, the symbols of URN's notation together with the term derived from meta-class name and attributes are shown. The symbols are visually grouped. The upper group the symbols of GRL, and the lower half the symbols of UCM. Within the groups, the symbols are not sorted. Not included were `Responsibility` and `Satisfaction Level ‘‘exceeded’’`.

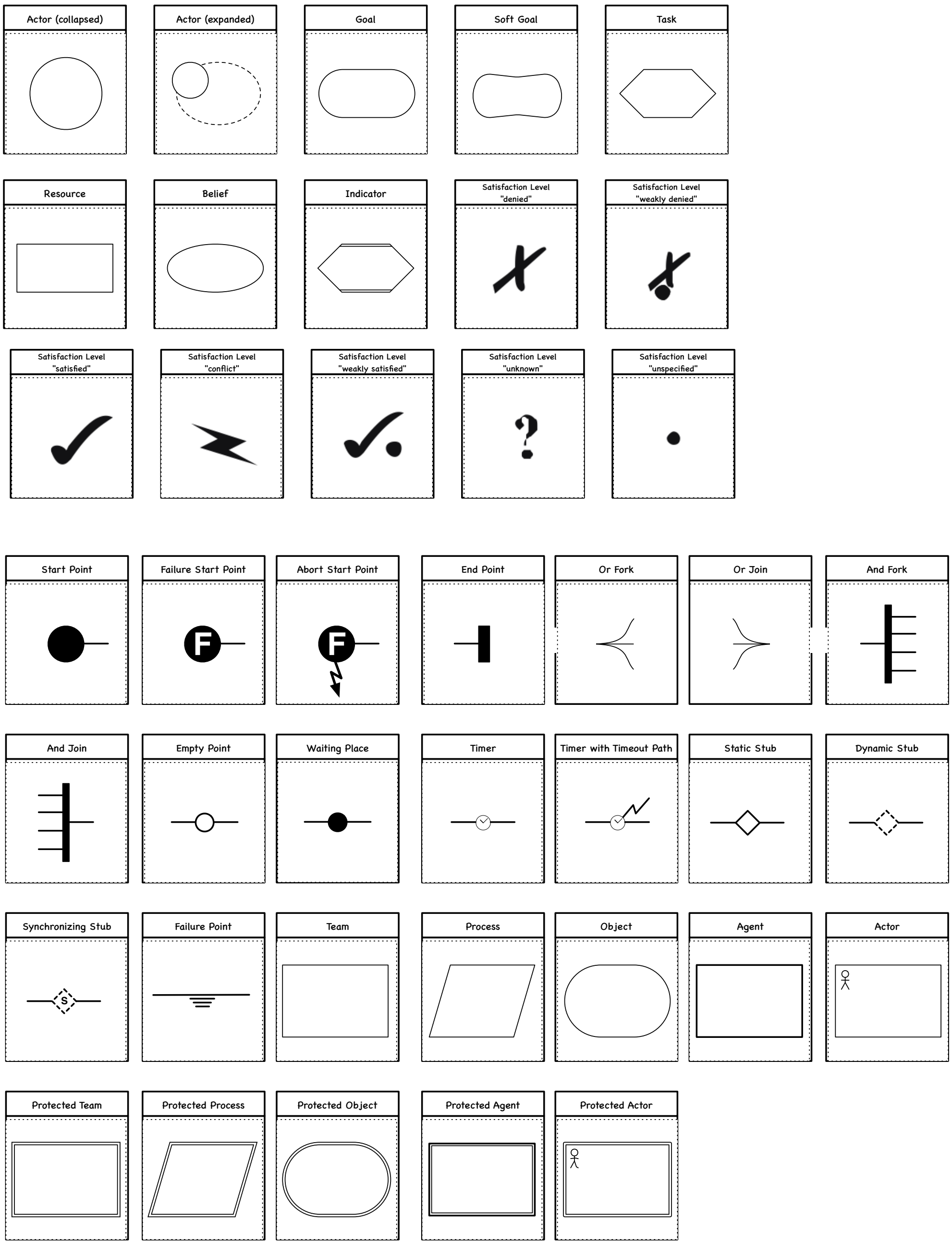


Figure A.169.: Notation tested in URN experiment

A.10.2. Errors Per Symbol

Table A.3.: Errors on URML concepts

Concept Name	#Total	# first	# second	# third	# fourth
System under Discussion	13	7	2	2	2
Procedure	12	6	1	1	4
Boundary Object	10	7	1	0	2
Assessment Sketch	10	6	1	1	2
Composite Entity Object	10	6	1	1	2
Functional Requirement	10	5	1	2	2
Stakeholder	10	4	1	2	3
Atomic Entity Object	9	7	1	0	1
Project Execution Quality Requirement	9	5	1	0	3
Leaf Environment Process	9	3	3	2	1
Regulatory Quality Requirement	8	4	2	1	1
Leaf Use Case	8	3	2	2	1
Environment Process	8	3	3	1	1
Regulatory Functional Requirement	7	5	1	0	1
Product Suite	7	4	2	1	0
Use Case	7	3	2	2	0
Under Construction Environment Process	7	2	3	1	1
Reliability Quality Requirement	7	4	2	1	0
Request	6	4	0	0	2
Functional Suitability Quality Requirement	6	4	1	0	1
Existing Product	6	4	1	1	0
Actor	6	3	1	1	1
System Service Provider	6	2	2	1	1
Service Provider	6	2	2	1	1
Product Line	5	4	0	0	1
Performance Quality Requirement	5	3	0	0	2

Table A.3.: Errors on URML concepts

Concept Name	#Total	# first	# second	# third	# fourth
Core Feature	5	3	0	1	1
Feature Group	5	2	1	1	1
Future Product	4	4	0	0	0
Hard Goal	4	4	0	0	0
Under Construction Feature Group	4	1	1	1	1
Under Construction Use Case	4	1	2	1	0
Soft Goal	3	3	0	0	0
Quality Requirement	3	2	1	0	0
Feature	3	1	0	1	1
Efficiency Quality Requirement	2	2	0	0	0
Hazard	2	2	0	0	0
Maintainability Quality Requirement	2	2	0	0	0
Mechanical Hazard	2	2	0	0	0
Social Hazard	2	2	0	0	0
Software Service Provider	2	2	0	0	0
Threat	2	2	0	0	0
Usability Quality Requirement	2	2	0	0	0
Business Stakeholder	2	1	0	0	1
Asset	2	1	1	0	0
Customer	2	1	1	0	0
Human Service Provider	2	1	1	0	0
Financial Asset	1	1	0	0	0
Idea	1	1	0	0	0
Identity Asset	1	1	0	0	0
Biological Hazard	1	1	0	0	0
Feature Tree	1	1	0	0	0
Meteorological Hazard	1	1	0	0	0
Radiological Hazard	1	1	0	0	0
Seismic Hazard	1	1	0	0	0

Table A.3.: Errors on URML concepts

Concept Name	**#Total**	**# first**	**# second**	**# third**	**# fourth**
Under Construction Feature Tree	1	1	0	0	0
Chemical Hazard	0	0	0	0	0
Electrical Hazard	0	0	0	0	0
Property Asset	0	0	0	0	0
Number of Errors	**275**	**160**	**45**	**29**	**41**

Table A.4.: Errors on KAOS concepts

Concept Name	**#Total**	**# first**	**# second**	**# third**	**# fourth**
Goal	15	7	2	1	5
Requirement	13	7	1	0	5
Object	13	6	1	1	5
Expectation	11	6	1	0	4
Entity	11	7	0	0	4
Operation	11	5	2	1	3
Soft Goal	12	7	1	1	3
State Machine	8	5	0	0	3
Model Annotation	9	5	1	1	2
Environment Agent	8	5	1	0	2
Software-To-Be Agent	8	5	1	0	2
State	8	4	1	1	2
Domain Property	9	7	0	0	2
Final State	6	3	2	0	1
Initial State	6	3	2	0	1
Event	8	6	1	0	1
Obstacle	5	4	0	0	1
Timeline	1	1	0	0	0
Number of Errors	**162**	**93**	**17**	**6**	**46**

Table A.5.: Errors on URN concepts

Concept Name	#Total	# first	# second	# third	# fourth
Team	16	7	4	0	5
Protected Team	15	6	4	0	5
Belief	15	7	3	1	4
Object	14	7	2	0	5
Protected Object	14	7	2	0	5
Indicator	14	7	3	0	4
Task	14	7	3	1	3
Failure Point	13	7	1	0	5
Agent	13	6	3	0	4
Protected Agent	13	6	3	0	4
Resource	12	6	2	1	3
Abort Start Point	11	7	0	0	4
Waiting Place	10	6	1	0	3
Static Stub	10	5	2	1	2
Protected Process	10	7	0	1	2
Process	10	7	0	1	2
Goal	10	5	3	0	2
Actor [collapsed]	10	7	1	0	2
Failure Start Point	9	5	0	0	4
Dynamic Stub	9	4	2	1	2
Soft Goal	9	5	2	0	2
Actor [expanded]	9	6	1	0	2
Satisfaction Level “unspecified”	8	4	1	0	3
And Join	8	4	1	1	2
End Point	8	5	1	0	2
Actor	8	5	1	0	2
Protected Actor	8	5	1	0	2
Or Fork	8	4	2	1	1
Synchronizing Stub	7	5	0	1	1
Or Join	6	3	1	1	1
Satisfaction Level “unknown”	5	1	1	0	3
And Fork	5	2	1	1	1
Timer with Timeout Path	4	2	1	1	0

Table A.5.: Errors on URN concepts

Concept Name	#Total	# first	# second	# third	# fourth
Timer	4	2	1	1	0
Start Point	2	2	0	0	0
Empty Point	1	1	0	0	0
Satisfaction Level “satisfied”	1	1	0	0	0
Satisfaction Level “weakly satisfied”	1	1	0	0	0
Satisfaction Level “conflict”	0	0	0	0	0
Satisfaction Level “denied”	0	0	0	0	0
Satisfaction Level “weakly denied”	0	0	0	0	0
Number of Errors	**344**	**184**	**54**	**14**	**92**

A.10.3. Erroneous Mappings

A.10.3.1. URML

The erroneous mappings listed here are already aggregated and sorted by the number of their occurrence. A given line of the form “N (Term:Symbol)” describes that Symbol has been mapped to term N times. After the mappings listing we present some additional graphical visualizations of mappings, that support the discussion in section 7.2.4.

5 (Assessment Sketch:Procedure)
4 (Regulatory Quality Requirement:Reliability Quality Requirement)
4 (Performance Quality Requirement:Project Execution Quality Requirement)
4 (Functional Requirement:Functional Suitability Quality Requirement)
3 (Under Construction Feature Group:Under Construction Environment Process)
3 (System under Discussion:Environment Process)
3 (System under Discussion:Boundary Object)
3 (System Service Provider:Service Provider)
3 (Stakeholder:Request)
3 (Stakeholder:Actor)
3 (Service Provider:System Service Provider)
3 (Request:Atomic Entity Object)
3 (Reliability Quality Requirement:Regulatory Quality Requirement)
3 (Project Execution Quality Requirement:Performance Quality Requirement)

3 (Procedure:Functional Requirement)
3 (Leaf Environment Process:Feature)
3 (Existing Product:Composite Entity Object)
3 (Environment Process:Use Case)
3 (Boundary Object:System under Discussion)
3 (Actor:Stakeholder)
2 (Use Case:Leaf Use Case)
2 (Use Case:Feature Group)
2 (Under Construction Use Case:Under Construction Feature Group)
2 (Under Construction Environment Process:Under Construction Use Case)
2 (Regulatory Functional Requirement:Regulatory Quality Requirement)
2 (Project Execution Quality Requirement:Assessment Sketch)
2 (Product Suite:System Service Provider)
2 (Product Line:Product Suite)
2 (Procedure:System under Discussion)
2 (Leaf Use Case:Core Feature)
2 (Leaf Environment Process:Leaf Use Case)
2 (Hard Goal:Assessment Sketch)
2 (Functional Suitability Quality Requirement:System under Discussion)
2 (Functional Suitability Quality Requirement:Project Execution Quality Requirement)
2 (Functional Requirement:Use Case)
2 (Feature Group:Environment Process)
2 (Environment Process:Leaf Environment Process)
2 (Core Feature:Existing Product)
2 (Composite Entity Object:Product Suite)
2 (Composite Entity Object:Product Line)
2 (Atomic Entity Object:Functional Requirement)
2 (Atomic Entity Object:Existing Product)
2 (Assessment Sketch:Soft Goal)
1 (Use Case:Under Construction Environment Process)
1 (Use Case:Environment Process)
1 (Use Case:Core Feature)
1 (Usability Quality Requirement:Seismic Hazard)
1 (Usability Quality Requirement:Procedure)
1 (Under Construction Use Case:Under Construction Environment Process)
1 (Under Construction Use Case:Boundary Object)
1 (Under Construction Feature Tree:Feature Tree)
1 (Under Construction Feature Group:Regulatory Quality Requirement)
1 (Under Construction Environment Process:Usability Quality Requirement)
1 (Under Construction Environment Process:Under Construction Feature Group)
1 (Under Construction Environment Process:Feature Group)
1 (Under Construction Environment Process:Asset)
1 (Under Construction Environment Process:Assessment Sketch)
1 (Threat:Mechanical Hazard)

1 (Threat:Boundary Object)
1 (System under Discussion:Under Construction Use Case)
1 (System under Discussion:Stakeholder)
1 (System under Discussion:Request)
1 (System under Discussion:Hard Goal)
1 (System under Discussion:Feature Group)
1 (System under Discussion:Assessment Sketch)
1 (System Service Provider:System under Discussion)
1 (System Service Provider:Procedure)
1 (System Service Provider:Actor)
1 (Stakeholder:Leaf Environment Process)
1 (Stakeholder:Human Service Provider)
1 (Stakeholder:Customer)
1 (Stakeholder:Core Asset)
1 (Software Service Provider:Under Construction Use Case)
1 (Software Service Provider:Assessment Sketch)
1 (Soft Goal:Hazard)
1 (Soft Goal:Hard Goal)
1 (Soft Goal:Assessment Sketch)
1 (Social Hazard:Hazard)
1 (Social Hazard:Composite Entity Object)
1 (Service Provider:Stakeholder)
1 (Service Provider:Quality Requirement)
1 (Service Provider:Atomic Entity Object)
1 (Seismic Hazard:Hard Goal)
1 (Request:Stakeholder)
1 (Request:Reliability Quality Requirement)
1 (Request:Assessment Sketch)
1 (Reliability Quality Requirement:Use Case)
1 (Reliability Quality Requirement:Under Construction Environment Process)
1 (Reliability Quality Requirement:Regulatory Functional Requirement)
1 (Regulatory Quality Requirement:System under Discussion)
1 (Regulatory Quality Requirement:Social Hazard)
1 (Regulatory Quality Requirement:Regulatory Functional Requirement)
1 (Regulatory Quality Requirement:Quality Requirement)
1 (Regulatory Quality Requirement:Environment Process)
1 (Regulatory Functional Requirement:Stakeholder)
1 (Regulatory Functional Requirement:Project Execution Quality Requirement)
1 (Regulatory Functional Requirement:Functional Suitability Quality Requirement)
1 (Regulatory Functional Requirement:Existing Product)
1 (Regulatory Functional Requirement:Boundary Object)
1 (Radiological Hazard:Biological Hazard)
1 (Quality Requirement:System under Discussion)
1 (Quality Requirement:Software Service Provider)
1 (Quality Requirement:Future Product)

1 (Project Execution Quality Requirement:Request)
1 (Project Execution Quality Requirement:Reliability Quality Requirement)
1 (Project Execution Quality Requirement:Regulatory Quality Requirement)
1 (Project Execution Quality Requirement:Procedure)
1 (Product Suite:Threat)
1 (Product Suite:Product Line)
1 (Product Suite:Leaf Use Case)
1 (Product Suite:Existing Product)
1 (Product Suite:Composite Entity Object)
1 (Product Line:Procedure)
1 (Product Line:Maintainability Quality Requirement)
1 (Product Line:Functional Requirement)
1 (Procedure:Request)
1 (Procedure:Human Service Provider)
1 (Procedure:Composite Entity Object)
1 (Procedure:Business Stakeholder)
1 (Procedure:Boundary Object)
1 (Procedure:Asset)
1 (Procedure:Assessment Sketch)
1 (Performance Quality Requirement:Mechanical Hazard)
1 (Meteorological Hazard:Soft Goal)
1 (Mechanical Hazard:Functional Requirement)
1 (Mechanical Hazard:Efficiency Quality Requirement)
1 (Maintainability Quality Requirement:Service Provider)
1 (Maintainability Quality Requirement:Atomic Entity Object)
1 (Leaf Use Case:Use Case)
1 (Leaf Use Case:System under Discussion)
1 (Leaf Use Case:Project Execution Quality Requirement)
1 (Leaf Use Case:Leaf Environment Process)
1 (Leaf Use Case:Future Product)
1 (Leaf Use Case:Atomic Entity Object)
1 (Leaf Environment Process:Under Construction Environment Process)
1 (Leaf Environment Process:Functional Requirement)
1 (Leaf Environment Process:Environment Process)
1 (Leaf Environment Process:Efficiency Quality Requirement)
1 (Identity Asset:Leaf Use Case)
1 (Idea:Identity Asset)
1 (Human Service Provider:Stakeholder)
1 (Human Service Provider:Actor)
1 (Hazard:Social Hazard)
1 (Hazard:Seismic Hazard)
1 (Hard Goal:Performance Quality Requirement)
1 (Hard Goal:Composite Entity Object)
1 (Future Product:Quality Requirement)
1 (Future Product:Idea)

1 (Future Product:Boundary Object)
1 (Future Product:Atomic Entity Object)
1 (Functional Suitability Quality Requirement:Stakeholder)
1 (Functional Suitability Quality Requirement:Product Line)
1 (Functional Requirement:Project Execution Quality Requirement)
1 (Functional Requirement:Product Suite)
1 (Functional Requirement:Portability Quality Requirement)
1 (Functional Requirement:Composite Entity Object)
1 (Financial Asset:Product Line)
1 (Feature:Regulatory Quality Requirement)
1 (Feature:Leaf Use Case)
1 (Feature:Leaf Environment Process)
1 (Feature Tree:Under Construction Feature Tree)
1 (Feature Group:Under Construction Feature Group)
1 (Feature Group:Service Provider)
1 (Feature Group:Leaf Environment Process)
1 (Existing Product:Reliability Quality Requirement)
1 (Existing Product:Regulatory Functional Requirement)
1 (Existing Product:Procedure)
1 (Environment Process:Procedure)
1 (Environment Process:Functionality Quality Requirement)
1 (Environment Process:Feature Group)
1 (Efficiency Quality Requirement:System under Discussion)
1 (Efficiency Quality Requirement:Maintainability Quality Requirement)
1 (Customer:Regulatory Functional Requirement)
1 (Customer:Actor)
1 (Core Feature:Leaf Environment Process)
1 (Core Feature:Hard Goal)
1 (Core Feature:Boundary Object)
1 (Composite Entity Object:Regulatory Functional Requirement)
1 (Composite Entity Object:Leaf Environment Process)
1 (Composite Entity Object:Future Product)
1 (Composite Entity Object:Functional Requirement)
1 (Composite Entity Object:Business Stakeholder)
1 (Composite Entity Object:Atomic Entity Object)
1 (Business Stakeholder:Stakeholder)
1 (Business Stakeholder:Leaf Use Case)
1 (Boundary Object:Usability Quality Requirement)
1 (Boundary Object:Service Provider)
1 (Boundary Object:Regulatory Functional Requirement)
1 (Boundary Object:Product Suite)
1 (Boundary Object:Future Product)
1 (Boundary Object:Composite Entity Object)
1 (Boundary Object:Atomic Entity Object)
1 (Biological Hazard:Radiological Hazard)

1 (Atomic Entity Object:System under Discussion)
1 (Atomic Entity Object:Software Service Provider)
1 (Atomic Entity Object:Regulatory Quality Requirement)
1 (Atomic Entity Object:Core Feature)
1 (Atomic Entity Object:Composite Entity Object)
1 (Asset:Product Suite)
1 (Asset:Financial Asset)
1 (Assessment Sketch:Threat)
1 (Assessment Sketch:Leaf Environment Process)
1 (Assessment Sketch:Functional Requirement)
1 (Actor:System Service Provider)
1 (Actor:Customer)
1 (Actor:Boundary Object)

A.10.3.2. KAOS

8 (Goal:Requirement)
7 (Object:Entity)
6 (Software-To-Be Agent:Environment Agent)
6 (Requirement:Expectation)
6 (Environment Agent:Software-To-Be Agent)
5 (Soft Goal:Goal)
4 (Soft Goal:Model Annotation)
4 (Initial State:Final State)
4 (Goal:Object)
4 (Final State:Initial State)
4 (Expectation:Soft Goal)
4 (Entity:State Machine)
4 (Entity:Object)
3 (State:Goal)
3 (Model Annotation:Soft Goal)
3 (Domain Property:Event)
2 (State Machine:Operation)
2 (Soft Goal:Initial State)
2 (Requirement:State Machine)
2 (Requirement:Goal)
2 (Operation:State)
2 (Operation:Expectation)
2 (Obstacle:Operation)
2 (Obstacle:Domain Property)
2 (Object:State Machine)
2 (Object:Operation)
2 (Model Annotation:Domain Property)
2 (Goal:Final State)
2 (Expectation:Goal)

2 (Event:Domain Property)
2 (Domain Property:State)
2 (Domain Property:Obstacle)
1 (Timeline:Software-To-Be Agent)
1 (State:Soft Goal)
1 (State:Requirement)
1 (State:Operation)
1 (State:Obstacle)
1 (State:Object)
1 (State Machine:State)
1 (State Machine:Software-To-Be Agent)
1 (State Machine:Soft Goal)
1 (State Machine:Requirement)
1 (State Machine:Expectation)
1 (State Machine:Entity)
1 (Software-To-Be Agent:Timeline)
1 (Software-To-Be Agent:Event)
1 (Soft Goal:Operation)
1 (Requirement:Soft Goal)
1 (Requirement:Event)
1 (Requirement:Domain Property)
1 (Operation:Requirement)
1 (Operation:Object)
1 (Operation:Model Annotation)
1 (Operation:Goal)
1 (Operation:Event)
1 (Operation:Entity)
1 (Operation:Domain Property)
1 (Obstacle:Soft Goal)
1 (Object:State)
1 (Object:Domain Property)
1 (Model Annotation:Operation)
1 (Model Annotation:Object)
1 (Model Annotation:Goal)
1 (Model Annotation:Entity)
1 (Initial State:Soft Goal)
1 (Initial State:Model Annotation)
1 (Goal:Environment Agent)
1 (Final State:Requirement)
1 (Final State:Goal)
1 (Expectation:State)
1 (Expectation:Operation)
1 (Expectation:Model Annotation)
1 (Expectation:Event)
1 (Expectation:Entity)

1 (Event:Requirement)
1 (Event:Operation)
1 (Event:Obstacle)
1 (Event:Object)
1 (Event:Expectation)
1 (Event:Environment Agent)
1 (Environment Agent:Obstacle)
1 (Environment Agent:Model Annotation)
1 (Entity:State)
1 (Entity:Model Annotation)
1 (Entity:Event)
1 (Domain Property:Object)
1 (Domain Property:Expectation)

A.10.3.3. URN

7 (Failure Start Point:Abort Start Point)
7 (Failure Point:Failure Start Point)
6 (Dynamic Stub:Static Stub)
5 (Static Stub:Synchronizing Stub)
5 (Satisfaction Level “unspecified”:Satisfaction Level “unknown”)
5 (Satisfaction Level “unknown”:Satisfaction Level “unspecified”)
5 (Protected Team:Protected Agent)
5 (Protected Object:Protected Team)
4 (Waiting Place:Team)
4 (Team:Agent)
4 (Task:Resource)
4 (Task:Belief)
4 (Protected Team:Protected Object)
4 (Protected Team:Protected Actor)
4 (Protected Agent:Protected Object)
4 (Or Join:Or Fork)
4 (Object:Team)
4 (Failure Point:Abort Start Point)
4 (Belief:Task)
3 (Team:Object)
3 (Team:Actor)
3 (Synchronizing Stub:Dynamic Stub)
3 (Protected Process:Protected Object)
3 (Protected Object:Protected Process)
3 (Protected Agent:Protected Team)
3 (Process:Object)
3 (Or Fork:And Join)
3 (Or Fork:And Fork)
3 (Object:Process)

3 (Indicator:Belief)
3 (End Point:Failure Point)
3 (Belief:Indicator)
3 (And Join:Or Join)
3 (And Join:Or Fork)
3 (And Fork:And Join)
3 (Agent:Team)
3 (Agent:Object)
3 (Actor:Agent)
3 (Actor [collapsed]:Soft Goal)
3 (Abort Start Point:Waiting Place)
2 (Timer:Timer with Timeout Path)
2 (Timer with Timeout Path:Timer)
2 (Timer with Timeout Path:Actor [expanded])
2 (Team:Protected Team)
2 (Static Stub:Dynamic Stub)
2 (Soft Goal:Actor [expanded])
2 (Resource:Team)
2 (Resource:Task)
2 (Resource:Indicator)
2 (Resource:Belief)
2 (Protected Process:Protected Agent)
2 (Protected Process:Indicator)
2 (Protected Object:Protected Agent)
2 (Protected Object:Indicator)
2 (Protected Agent:Protected Process)
2 (Protected Actor:Protected Team)
2 (Protected Actor:Protected Object)
2 (Protected Actor:Protected Agent)
2 (Process:Failure Point)
2 (Or Join:And Join)
2 (Or Fork:Or Join)
2 (Object:Task)
2 (Object:Soft Goal)
2 (Indicator:Task)
2 (Indicator:Failure Point)
2 (Indicator:Dynamic Stub)
2 (Indicator:Actor [collapsed])
2 (Goal:Dynamic Stub)
2 (Goal:Actor [collapsed])
2 (Failure Start Point:Failure Point)
2 (End Point:Start Point)
2 (Belief:Resource)
2 (Belief:Actor)
2 (And Join:And Fork)

2 (And Fork:Or Fork)
2 (Agent:Process)
2 (Actor [expanded]:Protected Process)
2 (Actor [collapsed]:Protected Agent)
2 (Abort Start Point:End Point)
1 (Waiting Place:Protected Agent)
1 (Waiting Place:Indicator)
1 (Waiting Place:Failure Point)
1 (Waiting Place:End Point)
1 (Waiting Place:Agent)
1 (Waiting Place:Actor [expanded])
1 (Timer:Belief)
1 (Timer:Actor [collapsed])
1 (Team:Resource)
1 (Team:Process)
1 (Team:Goal)
1 (Team:Actor [collapsed])
1 (Task:Waiting Place)
1 (Task:Timer with Timeout Path)
1 (Task:Object)
1 (Task:Indicator)
1 (Task:Goal)
1 (Task:Actor [collapsed])
1 (Synchronizing Stub:Static Stub)
1 (Synchronizing Stub:Resource)
1 (Synchronizing Stub:Failure Point)
1 (Synchronizing Stub:End Point)
1 (Static Stub:Resource)
1 (Static Stub:Empty Point)
1 (Static Stub:Actor [collapsed])
1 (Start Point:Waiting Place)
1 (Start Point:Actor [collapsed])
1 (Soft Goal:Timer with Timeout Path)
1 (Soft Goal:Team)
1 (Soft Goal:Synchronizing Stub)
1 (Soft Goal:Satisfaction Level “weakly satisfied”)
1 (Soft Goal:Protected Object)
1 (Soft Goal:Goal)
1 (Soft Goal:Failure Point)
1 (Satisfaction Level “weakly satisfied”:Goal)
1 (Satisfaction Level “unspecified”:Indicator)
1 (Satisfaction Level “unspecified”:End Point)
1 (Satisfaction Level “unspecified”:Belief)
1 (Satisfaction Level “satisfied”:Object)
1 (Resource:Failure Point)

1 (Resource:Empty Point)
1 (Resource:Agent)
1 (Resource:Actor [expanded])
1 (Protected Team:Synchronizing Stub)
1 (Protected Team:Actor [expanded])
1 (Protected Process:Process)
1 (Protected Process:Object)
1 (Protected Process:Actor [expanded])
1 (Protected Object:Goal)
1 (Protected Object:Belief)
1 (Protected Agent:Resource)
1 (Protected Agent:Protected Actor)
1 (Protected Agent:Indicator)
1 (Protected Agent:Actor [expanded])
1 (Protected Actor:Task)
1 (Protected Actor:Actor)
1 (Process:Team)
1 (Process:Task)
1 (Process:Protected Team)
1 (Process:Protected Process)
1 (Process:End Point)
1 (Object:Static Stub)
1 (Object:Resource)
1 (Object:Agent)
1 (Indicator:Satisfaction Level "unspecified")
1 (Indicator:Resource)
1 (Indicator:Actor)
1 (Goal:Timer)
1 (Goal:Soft Goal)
1 (Goal:Satisfaction Level "satisfied")
1 (Goal:Object)
1 (Goal:Belief)
1 (Goal:Agent)
1 (Failure Point:Waiting Place)
1 (Failure Point:End Point)
1 (End Point:Waiting Place)
1 (End Point:Satisfaction Level "unspecified")
1 (End Point:Failure Start Point)
1 (Empty Point:Goal)
1 (Dynamic Stub:Soft Goal)
1 (Dynamic Stub:Process)
1 (Dynamic Stub:Belief)
1 (Belief:Static Stub)
1 (Belief:Protected Object)
1 (Belief:Goal)

1 (Belief:Actor [expanded])
1 (Agent:Task)
1 (Agent:Protected Process)
1 (Agent:Indicator)
1 (Agent:Belief)
1 (Agent:Actor)
1 (Actor:Timer)
1 (Actor:Task)
1 (Actor:Protected Actor)
1 (Actor:Process)
1 (Actor:Goal)
1 (Actor [expanded]:Waiting Place)
1 (Actor [expanded]:Team)
1 (Actor [expanded]:Soft Goal)
1 (Actor [expanded]:Protected Team)
1 (Actor [expanded]:Object)
1 (Actor [expanded]:Goal)
1 (Actor [expanded]:Agent)
1 (Actor [collapsed]:Team)
1 (Actor [collapsed]:Protected Actor)
1 (Actor [collapsed]:Process)
1 (Actor [collapsed]:Goal)
1 (Actor [collapsed]:End Point)
1 (Abort Start Point:Static Stub)
1 (Abort Start Point:Soft Goal)
1 (Abort Start Point:Satisfaction Level “unspecified”)
1 (Abort Start Point:Protected Agent)
1 (Abort Start Point:Failure Start Point)
1 (Abort Start Point:Empty Point)

A.11. Definitions (Sources of Terminology)

Agent

(Yue 1986) ....an active component of a system. It can initiate events.

(Dardenne, Fickas, and Lamsweerde 1991, p.16) In KAOS, an agent is an object which is a processor for one action at least.

Actor

(Jacobson et al. 1992, p.127) The actors represent what interacts with the system. They represent everything that needs to exchange information with the system. Since the actors represent what is outside the system, we do not describe them in detail. Actors are not like other objects in the respect that their actions are non-deterministic.

(Rumbaugh, Jacobson, and Booch 2004) An actor is an idealization of a role played by an external person, process, or thing interacting with a system, subsystem, or class. An actor characterizes the interactions that a class of outside users may have with the system. At run time, one physical user may be bound to multiple actors within the system. Different users may be bound to the same actor and therefore represent multiple instances of the same actor definition. For example, one person may be a customer and a cashier of a store at different times. Each actor participates in one or more use cases. It interacts with the use case (and therefore with the system or class that owns the use case) by exchanging messages. The internal implementation of an actor is not relevant in the use case; an actor may be characterized sufficiently by a set of attributes that define its state. Actors may be defined in generalization hierarchies, in which an abstract actor description is shared and augmented by one or more specific actor descriptions. An actor may be a human, a computer system, or some executable process.

Boundary Object / Class

(Bruegge and Dutoit 2009, p.177) Boundary objects represent the interactions between the actors and the system.

Class

(p725) An abstraction of a set of objects with the same attributes, operations, relationships, and semantics. Classes are different than abstract data types in that a class can be defined by specializing another class.

(Jacobson et al. 1992, p.50) ... represents a template for several objects and descibes how these objects are stuctured internally. Objects of the same class have the same definition both for their operations and for their information structures.

Customer

(Gause and Weinberg 1989, p68) ...customers are the ones who pay us for the requirements work.

(Kang et al. 1990, p.3) The user of a system is not necessarily the same as the customer for a system. These are two separate concepts, although they may be combined in many cases.

Entity Object

(Jacobson et al. 1992, p.132) The entity object models information in the system that should be held for a longer time, and should typically survive a use case.

Feature

(Kang et al. 1990, p.8) A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems.

(p.28) ... capabilities from the perspective of end-users are modeled as features.

(p.35) Features are the attributes of a system that directly affect end-users.

(Kang et al. 1998, p.144) features are distinctively identifiable functional abstractions that must be implemented, tested, delivered, and maintained

(Lee, Kang, and Lee 2002, p.64) we use the term “features” as distinctive characteristics or properties of a product that differ from others or from earlier versions

(Kang, Lee, and Donohoe 2002) Customers and engineers usually speak of product characteristics in terms of the features the product has or delivers, so it’s natural and intuitive to express any commonality or variability in terms of features

Feature Tree (Pohl, Böckle, and Linden 2005, p.100) A feature tree hierarchically structures the set of features of a system. A feature can be decomposed into several sub-features that are mandatory, optional, or alternative.

Goal

(Yue 1987, p.43-44) In general, for a system, certain formulas and their denoted events can be identified as goals. In terms of the “causal chains”, the goals are the ends of the chains. They may enable something, but may not. [...] If the modeling primitives for a system [...] define a space of behaviors [Feather 87], then the goals of a system define a subset of that behavior space, namely, the set of behaviors in which the goals are satisfied.

(Johnson 1988, p.432) We call ... descriptions of desired behavior application goals.

(Dubois 1989, p.162) ...goals [...] are basic objectives and constraints that customers want to be met by the behavior resulting from the combination of the system and its environment ...

(Dardenne, Fickas, and Lamsweerde 1991, p.16) A goal is an objective that has to be met by the composite system. A goal is typically not formalized, in general, it cannot be described exclusively in terms of objects and actions of the composite system being considered. Hence, we say that the expression of a goal is nonoperational.

(Dardenne, Lamsweerde, and Fickas 1993, p.20) A goal is a nonoperational objective to be achieved by the composite system. Nonoperational means that the objective is not formulated in terms of objects and actions available to some agent in the system; in other words, a goal as it is formulated cannot be established through appropriate state transitions under control of one of the agents.

(ITU 2003) An objective or concern used to discover and evaluate functional and non-functional requirements.

Hard Goal

(ITU 2012, p.23) A (hard) Goal is a condition or state of affairs in the world that the stakeholders would like to achieve.

Soft Goal

(p.24) A Softgoal is a condition or state of affairs in the world that the actor would like to achieve, but unlike in the concept of (hard) goal, there are no clear-cut criteria for whether the condition is achieved, and it is up to subjective judgement and interpretation of the modeler to judge whether a particular state of affairs in fact achieves sufficiently the stated softgoal.

Interface Object / Class

(Jacobson et al. 1992, p.132-132) The interface object models behavior and information that is dependent on the interface to the system.

Model

(Jackson 1983, p.5) By 'model' we always mean a model of the real world outside of the computer system; we never mean a model of the system itself, nor of any of its attributes or characteristics, nor of the development procedure used to create it. Making a JSD model of the real world involves two distinct tasks: first, making an abstract description of the real world, and second, making a realization, in the computer, of that abstract description. The realization in the computer is then a model of the real world [...].

(p.376) An abstraction which has been realized, especially by a set of sequential processes.

(Yourdon 1988, p.65) Why should we build models? Why not just build the system itself? The answer is that we can construct models in such a way as to highlight, or emphasize, certain critical features of the system, while simultaneously de-emphasizing other aspects of the system.

(Davis 1993, p.369) A model simply provides us with a richer, higher level, and more semantically precise set of constructs than the underlying natural language. Using such a model reduces ambiguity, makes it easier to check for incompleteness, and may at times improve understandability. [...] as a model becomes increasingly complex, the model itself becomes increasingly more difficult to understand, independent of understanding what is being specified by that model.

(p.369) M is a model of a system S if M can be used to answer a well-defined set of questions about S to a tolerance adequate for a stated purpose.

(Martin and Odell 1998, p.355) An abstraction device. It expresses our concepts.

(Jackson 2001, p.12) First, a model can be just a useful description. An economic model may be a description of a country's economy: the model is a set of differential equations describing the relationships among prices, wages, unemployment nad inflation. Similarly, a model of a piece of computer software may be a description of its behavior as a finite-state machine relating external event stimuli to changes in the internal software state. In both cases the description is useful because it lets you analyse the possible behaviour of what it describes. These descriptions are called analytic models.

In the second meaning, a model is not a description but another reality with some similar properties. This is an analogic model.

(Lamsweerde 2009) A model is an abstract representation of the target system, where key features are highlighted, specified and inter-related to each other.

(Bruegge and Dutoit 2009, p.736) An abstraction of a system aimed at simplifying the reasoning about the system by omitting irrelevant details.

(ISO 2010, p.221) 1. a representation of a real world process, device, or concept [...] 2. a representation of something that suppresses certain aspects of the modeled subject [...] 3. an interpretation of a theory for which all the axioms of the theory are true [...] 4. a related collection of instances of meta-objects, representing (describing or prescribing) an information system, or parts thereof, such as a software product [...] 5. a semantically closed abstraction of a system or a complete description of a system from a particular perspective [...]

(Glinz 2014, p.15) An abstract representation of an existing reality or a reality to be created. [...] More generally speaking, a model is an abstract representation of an existing entity or an entity to be created, where entity denotes any part of reality or any other conceivable set of elements or phenomena, including other models. With respect to a model, the entity is called the original.

(Guide to the Software Engineering Body of Knowledge, Version 3.0 2014, p.163) A model is an abstraction or simplification of a software component.

(OMG 2015c, p.12) A model is always a model of something. The thing being modeled can generically be considered a system within some domain of discourse. The model then makes some statements of interest about that system, abstracting from all the details of the system that could possibly be described, from a certain point of view and for a certain purpose. For an existing system, the model may represent an analysis of the properties and behavior of that system. For a planned system, the model may represent a specification of how the system is to be constructed and behave.

Object

(Bruegge and Dutoit 2009, p.738) An instance of a class. An object has an identity and stores attribute values.

Obstacle

(Yue 1987, 44) In terms of goal satisfaction, some parts of the theory make positive contributions [...]. Inversely, if some parts make only negative contributions, they are “obstacles” (the use of this term is Don Cohen’s suggestion).

Requirement

(Bell and Thayer 1976, p.62) [...] software requirements describe functions that the software must perform, but not how they must be implemented.

(Kovitz 1999, p.35) Requirements are the effects that the computer is to exert in the problem domain, by virtue of the computer’s programming.

(Kotonya and Sommerville 1998, pp.3,4,6)[...] system requirements define what the system is required to do and the circumstances under which it is required to operate. [...] the requirements define the services that the system should provide and they set out constraints on the system’s operation. [...] Requirements are defined [...] as a specification of what should be implemented. They are descriptions of how the system should behave, application domain information, constraints on the system’s operation, or specifications of a system property or attribute. Sometimes they are constraints on the development process of the system. [...] What are requirements? A statement of a system service or constraint

Requirements Model

(Jacobson et al. 1992, p.123) From the requirements specification, a requirements model is created in which we specify all the functionality of the system. This mainly done by use cases in the use case model which is a part of the requirement model. [...] The requirements model also forms the basis of another model created by the analysis process, namely the analysis model.

(p.126) The requirements model consists of: • A use case model • Interface descriptions • A problem domain model

(p.156) The requirements model aims to delimit the system and define the functionality that the system should offer. This model could function as a contract between the developer and the orderer of the system and thus forms the developer’s view of what the customer wants. Therefore it is essential that non-OOSE practitioners should also be able to read this model. The requirements model will govern the development of all the other models, so this model is central throughout the whole system development.

(Glinz 2014, p.18) A model that has been created with the purpose of specifying requirements.

Role

(Jacobson 1987, 186) A role is defined through a specified task or a group of closely related tasks which are performed by persons during the development and/or operation of a system.

Stakeholder

(Kotonya and Sommerville 1998, p.10) System stakeholders are people or organizations who will be affected by the system and who have a direct or indirect influence on the system requirements. They include end-users of the system, managers and others involved in the organizational processes influenced by the system, engineers responsible for the system development and maintenance, customers of the organization who will use the system to provide some services, external bodies such as regulators or certification authorities [...]

(Alexander and Robertson 2004, p.23) A project stakeholder is someone who gains or loses something (could be functionality, revenue, status, compliance with rules, and so on) as a result of that project.

Threat

(ISO 2010) A threat is defined as "a state of the system or system environment which can lead to adverse effect in one or more given risk dimensions" .

System

(Feather 1987) ...composite systems, by which I mean systems whose realization will be multiple interacting components.

Use Case

(Jacobson 1987, pp. 185, 186) ... different aspects, each corresponding to a behaviorally related sequence, are called use cases. [...] we want to structure the system's total behavior in aspects, where each aspect corresponds to what we can call a use case. [...] A use case is a special sequence of transactions, performed by a user and a system in a dialogue. A transaction is performed by either the user or the system and is a response initiated by a stimulus. When no more stimuli can be generated, all the transactions will finally ebb away. The use case is ended and a new use case can be initiated. [...] Each use case is described in a semi-formal manner by using structured English or by using graphics for example a data flow chart as suggested by Gane and Sarson [l0].

(Jacobson et al. 1992, p.127) When a user uses the system, she or he will perform a behaviorally related sequence of transactions in a dialogue with the system. We call such a special sequence a use case. [...] Each use case is a specific way of using the system and every execution of the use case may be viewed as an instance of the use case.

(Rumbaugh, Jacobson, and Booch 2004) A use case is a coherent unit of externally visible functionality provided by a classifier (called the subject) and expressed by sequences of messages exchanged by the subject and one or more actors of the system unit. The purpose of a use case is to define a piece of coherent behavior without revealing the internal structure of the subject. The definition of a use case includes all the behavior it entails—the mainline sequences, different variations on normal behavior, and all the exceptional conditions that can occur with such behavior, together with the desired response. From the user's point of view, these may be abnormal situations. From the system's point of view, they are additional variations that must be described and handled.

User

(Jacobson 1987, 186) A user is a specific type of role, namely a role that participates in the operation of the system.

(Jacobson et al. 1992, p.127) We differentiate between actors and users. The user is the actual person who uses the system, whereas an actor represents a certain role that a user can play. We regard an actor as a class and users as instances of this class.

(Gause and Weinberg 1989, pp68-69) A user is any individual who is affected by, or affects, the product being designed. [...] Customers are obviously users, too, because paying for the product certainly must affect their bank accounts. But not all users are customers. [...] The users are the people that make or break the product.

Terminology conflicts

(Leite 1988, p.2) [...] research labeled as dealing with requirements, deals with specification, and this is the main reason for the lack of agreement on the definitions of requirements analysis and specification.

(Davis 1990, p.23) It is remarkable how inconsistent the terminology used in the software industry is. [...] there is no consistent use of the terms requirements, analysis, specification, and design.

(Mylopoulos, Chung, and Nixon 1992, p.1) [...] non-functional requirements (also referred to as constraints, goals, and quality attributes in the literature) ... [footnote put in parentheses]

(Letier 2001, p.17) In order to keep a uniform terminology throughout the thesis, we changed the terminology used by Jackson and Zave. In their terminology, a goal is called a requirement, and a requirement is called a specification.

A.12. Galileo Report Data Visualization

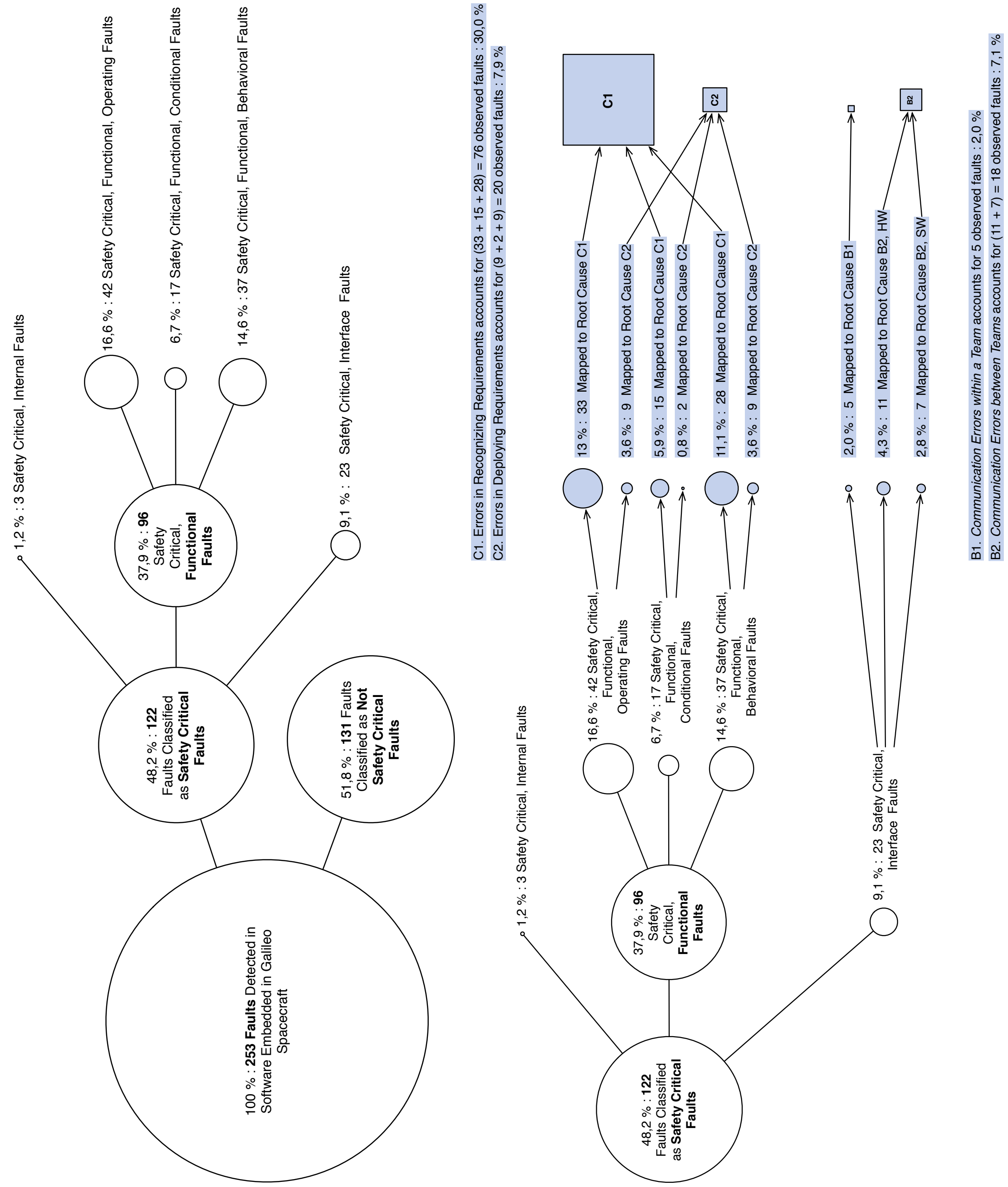

100 % : 253 Faults Detected in Software Embedded in Galileo Spacecraft
48,2 % : 122 Faults Classified as Safety Critical Faults
51,8 % : 131 Faults Classified as Not Safety Critical Faults
1,2 % : 3 Safety Critical, Internal Faults
37,9 % : 96 Safety Critical, Functional Faults
9,1 % : 23 Safety Critical, Interface Faults
16,6 % : 42 Safety Critical, Functional, Operating Faults
6,7 % : 17 Safety Critical, Functional, Conditional Faults
14,6 % : 37 Safety Critical, Functional, Behavioral Faults
48,2 % : 122 Faults Classified as Safety Critical Faults
1,2 % : 3 Safety Critical, Internal Faults
37,9 % : 96 Safety Critical, Functional Faults
9,1 % : 23 Safety Critical, Interface Faults
16,6 % : 42 Safety Critical, Functional, Operating Faults
6,7 % : 17 Safety Critical, Functional, Conditional Faults
14,6 % : 37 Safety Critical, Functional, Behavioral Faults
13 % : 33 Mapped to Root Cause C1
3,6 % : 9 Mapped to Root Cause C2
5,9 % : 15 Mapped to Root Cause C1
0,8 % : 2 Mapped to Root Cause C2
11,1 % : 28 Mapped to Root Cause C1
3,6 % : 9 Mapped to Root Cause C2
2,0 % : 5 Mapped to Root Cause B1
4,3 % : 11 Mapped to Root Cause B2, HW
2,8 % : 7 Mapped to Root Cause B2, SW
C1
C2
B2
C1. Errors in Recognizing Requirements accounts for (33 + 15 + 28) = 76 observed faults : 30,0 %
C2. Errors in Deploying Requirements accounts for (9 + 2 + 9) = 20 observed faults : 7,9 %
B1. Communication Errors within a Team accounts for 5 observed faults : 2,0 %
B2. Communication Errors between Teams accounts for (11 + 7) = 18 observed faults : 7,1 %

Figure A.170.: Data taken from (Lutz 1992)

Bibliography

Alexander, I, and S Robertson. 2004. “Requirements - Understanding project sociology by modeling stakeholders.” *IEEE Software* 21, no. 1 (January): 23–27.

Amyot, Daniel, and Gunter Mussbacher. 2011. “User Requirements Notation: The First Ten Years, The Next Ten Years (Invited Paper).” *Journal of Software* 6, no. 5 (May).

Baar, Thomas. 2006. “Correctly Defined Concrete Syntax for Visual Modeling Languages.” In *Model Driven Engineering Languages and Systems: 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006. Proceedings,* edited by Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, 111–125. Berlin, Heidelberg: Springer Berlin Heidelberg.

Beckenstein, M. 1972. “System requirements analysis: A management tool.” *IEEE Transactions on Engineering Management* EM-19, no. 4 (November): 124–128.

Bell, T. E., and T. A. Thayer. 1976. “Software requirements: Are they really a problem?” In *2nd international conference on Software engineering,* 61–68. San Francisco, California, United States: IEEE Computer Society Press.

Bell, Thomas E., and David C. Bixler. 1976. “A flow oriented requirements statement language.” In *Proceedings of the Symposium on Computer Software Engineering,* 109–122. New York: Polytechnic Press.

Berenbach, Brian, Daniel J Paulish, Juergen Kazmeier, and Arnold Rudorfer. 2009. *Software & systems requirements engineering : in practice.* New York : McGraw-Hill.

Berenbach, Brian, Diane Rea, and Florian Schneider. 2013. “Process Modeling for Requirements Engineering: A Medical System Case Study.” In *INCOSE International Symposium (IS) 2013,* 319–330.

Berenbach, Brian, Florian Schneider, and Helmut Naughton. 2012. “The use of a requirements modeling language for industrial applications.” In *Requirements Engineering Conference (RE), 2012 20th IEEE International.*

Bertin, Jacques. 1967. *Semiologie graphique : Les diagrammes, les réseaux, les cartes.* Paris: Mouton.

Bertin, Jacques, and William J Berg. 2011. *Semiology of graphics : diagrams, networks, maps.* Redlands, Calif.: ESRI Press : Distributed by Ingram Publisher Services.

Bleistein, Stephen J, Karl Cox, June Verner, and Keith T Phalp. 2006. "B-SCP: A requirements analysis framework for validating strategic alignment of organizational IT based on strategy, context, and process." *Information and Software Technology* 48 (9): 846–868.

Braber, F den, I Hogganvik, M S Lund, K Stølen, and F Vraalsen. 2007. "Model-based security analysis in seven steps — a guided tour to the CORAS method." *BT Technology Journal* 25, no. 1 (January): 101–117.

Broy, Manfred. 2010. "Multifunctional software systems: Structured modeling and specification of functional requirements." *Special Section on the Programming Languages Track at the 23rd ACM Symposium on Applied Computing ACM SAC 08* 75 (12): 1193–1214.

Bruegge, Bernd, and Allen H Dutoit. 2009. *Object-Oriented Software Engineering Using UML, Patterns, and Java.* 3rd ed. Prentice Hall, August.

Buhr, R J A, and R S Casselman. 1995. *Use case maps for object-oriented systems.* Prentice-Hall, Inc., November.

Cailliau, Antoine, C Damas, B Lambeau, and Axel van Lamsweerde. 2013a. "Modeling car crash management with KAOS." In *Comparing Requirements Modeling Approaches Workshop (CMA@RE), 2013 International,* 19–24.

Cailliau, Antoine, Christophe Damas, Bernard Lambeau, and Axel van Lamsweerde. 2013b. *Modeling Car Crash Management with KAOS.* Technical report. Université catholique de Louvain.

Caire, P, N Genon, P Heymans, and D L Moody. 2013. "Visual notation design 2.0: Towards user comprehensible requirements engineering notations." In *Requirements Engineering Conference (RE), 2013 21st IEEE International,* 115–124. July.

Capozucca, Alfredo, Betty H C Cheng, Geri Georg, Nicolas Guelfi, Paul Istoan, and Gunter Mussbacher. 2012. *Requirements Definition Document for a Software Product Line of Car Crash Management Systems.* University of Ottawa, May.

Carroll, John M. 2000. *Making use : scenario-based design of human-computer interactions.* Cambridge, Mass.: MIT Press.

Castro, Jaelson, Manuel Kolp, and John Mylopoulos. 2001. "A Requirements-Driven Development Methodology." In *Advanced Information Systems Engineering: 13th International Conference, CAiSE 2001 Interlaken, Switzerland, June 4–8, 2001 Proceedings,* edited by Klaus R Dittrich, Andreas Geppert, and Moira C Norrie, 108–123. Berlin, Heidelberg: Springer Berlin Heidelberg.

Chung, Lawrence, and Julio Cesar Sampaio do Prado Leite. 2009. "On Non-Functional Requirements in Software Engineering." In *Conceptual Modeling: Foundations and Applications,* edited by Alexander Borgida, Vinay Chaudhri, Paolo Giorgini, and Eric Yu, 363–379. Springer Berlin / Heidelberg.

Chung, Lawrence, Brian A Nixon, Eric Yu, and John Mylopoulos. 2000. *Non-Functional Requirements in Software Engineering.* Boston, MA: Springer US.

Class, Jakob. 2012. "A UML Profile for a new Requirements Modeling Language." Master's thesis, Technische Universität München.

Classen, Andreas, Patrick Heymans, and Pierre-Yves Schobbens. 2008. "What's in a Feature: A Requirements Engineering Perspective." In *Fundamental Approaches to Software Engineering,* edited by José Fiadeiro and Paola Inverardi, 16–30. Springer Berlin / Heidelberg.

CMMI Product Team. 2010. *CMMI for Development, Version 1.3.* Technical report. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.

Cockburn, Alistair. 2000. *Writing Effective Use Cases.* 1st ed. Addison-Wesley Professional, October.

Cooper, Alan. 1999. *The inmates are running the asylum.* Indianapolis, IN: Sams.

Dano, B, H Briand, and F Barbier. 1997. "A use case driven requirements engineering process." *Requirements Engineering* 2, no. 2 (June): 79–91.

Dardenne, A, Stephen Frank Fickas, and Axel van Lamsweerde. 1991. "Goal-directed concept acquisition in requirements elicitation." In *Sixth International Workshop on Software Specification and Design,* 14–21. IEEE Comput. Soc. Press.

Dardenne, Anne, Axel van Lamsweerde, and Stephen Frank Fickas. 1993. "Goal-directed requirements acquisition." *Science of computer programming* 20 (1-2): 3–50.

Darimont, Robert, and Axel van Lamsweerde. 1996. "Formal refinement patterns for goal-driven requirements elaboration." *SIGSOFT Softw. Eng. Notes* 21 (6): 179–190.

Davis, Alan Mark. 1990. *Software requirements : analysis and specification.* Englewood Cliffs, N.J.: Prentice Hall.

———. 1993. *Software requirements : objects, functions, and states.* Englewood Cliffs, N.J.: PTR Prentice Hall.

Denger, C, Daniel M Berry, and E Kamsties. 2003. "Higher quality requirements specifications through natural language patterns." In *Software: Science, Technology and Engineering, 2003. SwSTE '03. Proceedings. IEEE International Conference on,* 80–90. IEEE Comput. Soc.

Dijkstra, Edsger W. 1972. "The humble programmer." *Communications of the ACM* 15, no. 10 (October): 859–866.

DIN 4844-2 (2011). 2010. Deutsches Institut für Normung e.V. December.

Dubois, E, J Hagelstein, E Lahou, F Ponsaert, and A Rifaut. 1986. "A knowledge representation language for requirements engineering." *Proceedings of the IEEE* 74 (10): 1431–1444.

Dubois, Eric. 1989. "A logic of action for supporting goal-oriented elaborations of requirements." In *IWSSD '89: Proceedings of the 5th international workshop on Software specification and design.* ACM, April.

Ehn, Pelle. 1988. "Work-Oriented Design of Computer Artifacts." PhD diss., Department of information Processing, Umeå University.

Eichelberger, Holger, Yilmaz Eldogan, and Klaus Schmid. 2009. "A Comprehensive Survey of UML Compliance in Current Modelling Tools." In *Software Engineering 2009,* edited by Peter Liggesmeyer, Gregor Engels, Jürgen Münch, Jörg Dörr, and Norman Riegel, 39–50. Bonn, Germany: Gesellschaft für Informatik e.V. (GI), January.

Elahi, Golnaz, Eric Yu, and Nicola Zannone. 2009. "A vulnerability-centric requirements engineering framework: analyzing security attacks, countermeasures, and requirements based on vulnerabilities." *Requirements Engineering* 15, no. 1 (November): 41–62.

Feather, Martin S. 1987. "Language support for the specification and development of composite systems." *Transactions on Programming Languages and Systems (TOPLAS* 9, no. 2 (March).

"feature." 2011. Accessed June 9, 2016. `https://ahdictionary.com/word/search.html?q=feature`.

Figl, Kathrin, Michael Derntl, Manuel Caeiro Rodriguez, and Luca Botturi. 2010. "Cognitive effectiveness of visual instructional design languages." *Journal of Visual Languages and Computing* 21, no. 6 (December): 359–373.

Fondement, Frédéric, and Thomas Baar. 2005. "Making Metamodels Aware of Concrete Syntax." In *Model Driven Architecture – Foundations and Applications,* 190–204. Berlin, Heidelberg: Springer Berlin Heidelberg.

Fowler, M. 2004. *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* 3rd. Addison-Wesley Object Technology Series. Addison-Wesley.

Fowler, Martin, and Kent Beck. 1999. *Refactoring : improving the design of existing code.* Reading, MA: Addison-Wesley.

Gamma, Erich, Richard Helm, Ralph E Johnson, and John M Vlissides. 1995. *Design patterns : elements of reusable object-oriented software.* Reading, Mass.: Addison-Wesley.

Gause, D C, and G M Weinberg. 1989. *Exploring requirements: quality before design.* Dorset House Pub.

Genon, Nicolas, Daniel Amyot, and Patrick Heymans. 2011. "Analysing the Cognitive Effectiveness of the UCM Visual Notation." In *System Analysis and Modeling: About Models,* edited by Frank Alexander Kraemer and Peter Herrmann, 221–240. Berlin, Heidelberg: Springer Berlin Heidelberg.

Genon, Nicolas, Patrick Heymans, and Daniel Amyot. 2011. "Analysing the Cognitive Effectiveness of the BPMN 2.0 Visual Notation." In *Software Language Engineering. Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers,* 377–396. Berlin, Heidelberg: Springer Berlin Heidelberg.

Ghazi, Parisa, Norbert Seyff, and Martin Glinz. 2015. "FlexiView: A Magnet-Based Approach for Visualizing Requirements Artifacts." In *Requirements Engineering: Foundation for Software Quality,* 262–269. Cham: Springer International Publishing, March.

Glinz, Martin. 2000. “Problems and deficiencies of UML as a requirements specification language.” In *Software Specification and Design, 2000. Tenth International Workshop on,* 11–22.

———. 2010. “Very Lightweight Requirements Modeling.” In *Requirements Engineering Conference (RE), 2010 18th IEEE International,* 385–386.

———. 2014. *A Glossary of Requirements Engineering Terminology.* International Requirements Engineering Board (IREB), May.

Glinz, Martin, Stefan Berner, and Stefan Joos. 2002. “Object-oriented modeling with Adora.” *Information Systems* 27, no. 6 (September): 425–444.

Glinz, Martin, and Dustin Wüest. 2010. *A Vision of an Ultralightweight Requirements Modeling Language.* Technical report 2010.IFI-2010.0006. Zürich, Switzerland: University of Zurich, Department of Informatics, July.

Gotel, Orlena C Z, and Jane Cleland-Huang. 2009. “Next Top Model: A Requirements Engineering Reality Panel.” In *Requirements Engineering Conference, 2009. RE ’09. 17th IEEE International,* 357.

Green, T R G, and M Petre. 1996. “Usability Analysis of Visual Programming Environments: A ‘Cognitive Dimensions’ Framework.” *Journal of Visual Languages and Computing* 7, no. 2 (June): 131–174.

Greenspan, Sol J, John Mylopoulos, and Alex Borgida. 1982. “Capturing more world knowledge in the requirements specification.” In *ICSE ’82: Proceedings of the 6th international conference on Software engineering.* IEEE Computer Society Press, September.

Grundy, John C, John Hosking, Karen Na Li, Norhayati Mohd Ali, Jun Huh, and Richard Lei Li. 2013. “Generating Domain-Specific Visual Language Tools from Abstract Visual Specifications.” *IEEE Transactions on Software Engineering* 39, no. 4 (April): 487–515.

Guide to the Software Engineering Body of Knowledge, Version 3.0. 2014. IEEE Computer Society, August.

Gunter, Carl A, Elsa L Gunter, Michael A Jackson, and Pamela Zave. 2000. “A reference model for requirements and specifications.” *IEEE Software* 17, no. 3 (May): 37–43.

Haeger, Michael. 2011. “A Unified Requirements Model for System Engineering.” PhD diss., Technische Universität München.

Harel, David. 1987. “Statecharts: a visual formalism for complex systems.” *Science of computer programming* 8 (3): 231–274.

Helming, Jonas, Maximilian Kögel, Bernd Bruegge, and Brian Berenbach. 2010. “Unified Requirements Modeling for Environmental Systems.” In *Second International Workshop on Software Research and Climate Change,* 3–4. Cape Town, South Africa.

Helming, Jonas, Maximilian Kögel, Florian Schneider, M Haeger, C Kaminski, Bernd Bruegge, and Brian Berenbach. 2010. “Towards a unified Requirements Modeling Language.” In *Requirements Engineering Visualization, 2010 Fifth International Workshop on,* 53–57.

“hemolysis.” 2016. Accessed June 20, 2016. `http://www.dictionary.com/browse/hemolysis`.

Höcht, Sebastian. 2012. “A comparative analysis of Goal-Oriented Requirements Engineering approaches.” Master’s thesis, Technische Universität München.

Horkoff, Jennifer. 2012. “Comparing Modeling Approaches Workshop 2012” (September): 1–50.

Howell, William C, and Alfred H Fuchs. 1968. “Population stereotypy in code design.” *Organizational Behavior and Human Performance* 3, no. 3 (August): 310–339.

Hsia, P, A.M. Davis, and D C Kung. 1993. “Status report: requirements engineering.” *IEEE Software* 10 (6): 75–79.

IEEE. 2005. *1012 - IEEE Standard for Software Verification and Validation.* IEEE Computer Society.

ISO (International Organization for Standardization). 2011. *ISO 7010.* International Organization for Standardization (ISO), June.

ISO (International Organization for Standardization and International Electrotechnical Commission). 2001. *ISO/IEC 9126-1 - Software engineering – Product quality – Part 1: Quality model.* ISO (International Organization for Standardization).

ISO (International Organization for Standardization and International Electrotechnical Commission). 2005. *ISO/IEC 25000. Software engineering — Software product Quality Requirements and Evaluation (SQuaRE) — Guide to SQuaRE.* ISO/IEC, July.

ISO (International Organization for Standardization and International Electrotechnical Commission). 2011. *ISO/IEC 25010 - Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models.* ISO/IEC, March.

ISO (International Organization for Standardization, International Electrotechnical Commission, and Institute of Electrical and Electronics Engineers). 2010. *ISO/IEC/IEEE 24765:2010 - Systems and software engineering – Vocabulary.* ISO/IEC/IEEE, December.

ITU (International Telecommunication Union). 2003. *ITU-T Z.150: User Requirements Notation (URN) – Language requirements and framework.* International Telecommunication Union.

———. 2008a. *ITU-T Z.151: User requirements notation (URN) — Language definition,* November.

———. 2008b. *Notations and guidelines for the definition of ITU-T languages,* November.

———. 2011. *Message Sequence Chart (MSC),* August.

———. 2012. *ITU-T Z.151: User requirements notation (URN) – Language definition.* International Telecommunication Union.

Jackson, M A. 1983. *System development.* Englewood Cliffs, N.J: Prentice/Hall.

———. 2001. *Problem frames : analysing and structuring software development problems.* Harlow, England; New York: Addison-Wesley/ACM Press.

Jackson, Michael A, and Pamela Zave. 1993. “Domain descriptions.” In *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on,* 56–64.

———. 1995. “Deriving Specifications from Requirements: an Example.” In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on,* 15–24.

Jacobson, Ivar. 1987. “Object-oriented development in an industrial environment.” In *OOPSLA ’87: Conference proceedings on Object-oriented programming systems, languages and applications.* December.

Jacobson, Ivar, Grady Booch, and James Rumbaugh. 1999. *The unified software development process.* Reading, Mass: Addison-Wesley.

Jacobson, Ivar, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. 1992. *Object-oriented software engineering: a use case driven approach.* Addison-Wesley.

Jarke, Matthias. 1999. "Crews: Towards Systematic Usage of Scenarios, Use Cases and Scenes." In *Electronic Business Engineering: 4.Internationale Tagung Wirtschaftsinformatik 1999,* edited by Markus Nüttgens and August-Wilhelm Scheer, 469–486. Heidelberg: Physica-Verlag HD.

Johnson, W L. 1988. *Deriving specifications from requirements.* IEEE Computer Society Press, April.

Jones, Sheila. 1983. "Stereotypy in pictograms of abstract concepts." *Ergonomics* 26, no. 6 (June): 605–611.

Joos, Stefan. 2000. "ADORA-L – Eine Modellierungssprache zur Spezifikation von Software-Anforderungen." PhD diss., Wirtschaftswissenschaftliche Fakultät der Universität Zürich.

"jUCMNav: Eclipse plugin for the User Requirements Notation." 2001. Accessed February 16, 2016. `http://jucmnav.softwareengineering.ca/ucm/bin/view/ProjetSEG/WebHome`.

Kaminski, Christine. 2011. "A Comprehensible Visualization of the Unified Requirements Modeling Language." Master's thesis, Technische Universität München.

Kang, Kyo C, Sholom G Cohen, James A Hess, William E Nova, and A Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical report CMU/SEI-90-TR-21. Pittsburgh: Software Engineering Institute, Carnegie Mellon University.

Kang, Kyo C, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. 1998. "FORM: A feature-oriented reuse method with domain-specific reference architectures." *Annals of Software Engineering* 5 (1): 143–168.

Kang, Kyo C, Jaejoon Lee, and Patrick Donohoe. 2002. "Feature-oriented product line engineering." *IEEE Software* 19 (4): 58–65.

Kelly, Steven. 1997. "Towards a Comprehensive MetaCASE and CAME Environment. Conceptual, Architectural, Functional and Usability Advances in MetaEdit+." PhD diss., UNIVERSITY OF JYVÄSKYLÄ.

Kelly, Steven, and Juha-Pekka Tolvanen. 2007. "Appendix A: Metamodeling Language." In *Domain-Specific Modeling,* 411–414. Hoboken, NJ, USA: John Wiley & Sons, Inc.

Kleppe, Anneke. 2008. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels.* 1st ed. Addison-Wesley Professional, December.

Kotonya, Gerald, and Ian Sommerville. 1998. *Requirements Engineering: Processes and Techniques.* John Wiley & Sons, Inc., August.

Kovitz, Benjamin L. 1999. *Practical software requirements : a manual of content and style.* Greenwich, CT.: Manning.

Lamsweerde, Axel van. 2001. "Goal-oriented requirements engineering: a guided tour." In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on,* 249–262.

———. 2003. "Goal-oriented requirements engineering: from system objectives to UML models to precise software specifications." In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on,* 744–745. IEEE.

———. 2004. "Elaborating Security Requirements by Construction of Intentional Anti-Models." In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering.* IEEE Computer Society, May.

———. 2009. *Requirements Engineering - From System Goals to UML Models to Software Specifications.* Wiley.

Lamsweerde, Axel van, R Darimont, and Philippe Massonet. 1995. "Goal-directed elaboration of requirements for a meeting scheduler: problems and lessons learnt." In *Requirements Engineering, 1995., Proceedings of the Second IEEE International Symposium on,* 194–203. IEEE Comput. Soc. Press.

Lamsweerde, Axel van, and Emmanuel Letier. 1998. "Integrating obstacles in goal-driven requirements engineering." In *Software Engineering, 1998. Proceedings of the 1998 International Conference on,* 53–62.

———. 2000. "Handling obstacles in goal-oriented requirements engineering." *IEEE Transactions on Software Engineering* 26 (10): 978–1005.

Lee, Jaejoon, Kyo C Kang, Pete Sawyer, and Hyesun Lee. 2013. "A holistic approach to feature modeling for product line requirements engineering." *Requirements Engineering* 19, no. 4 (September): 377–395.

Lee, Kwanwoo, Kyo C Kang, and Jaejoon Lee. 2002. "Concepts and Guidelines of Feature Modeling for Product Line Software Engineering." In *Software Reuse: Methods, Techniques, and Tools,* edited by Cristina Gacek, 62–77. Springer Berlin / Heidelberg.

Leite, Julio Cesar Sampaio do Prado. 1988. "Viewpoint Resolution in Requirements Elicitation." PhD diss., University of Califonia Irvine.

Letier, Emmanuel. 2001. "Reasoning about Agents in Goal-Oriented Requirements Engineering." PhD diss., Université Catholique du Louvain (UCL), Département d'Ingénierie Informatique.

Letier, Emmanuel, and Axel van Lamsweerde. 2002. "Agent-based tactics for goal-oriented requirements elaboration." In *the 24th international conference,* 83. New York, New York, USA: ACM Press.

Leveson, Nancy G. 2000. "Intent Specifications: An Approach to Building Human-Centered Specifications." *IEEE Transactions on Software Engineering* 28, no. 1 (January): 15–35.

———. 2011. *Engineering a Safer World.* Engineering Systems. Cambridge, MA, USA: MIT Press.

Li, F L, Jennifer Horkoff, John Mylopoulos, L Liu, and A. Borgida. 2013. "Non-Functional Requirements Revisited." In *Proceedings of the 6th International i* Workshop (iStar 2013).* January.

Lutz, Robyn R. 1992. *Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems.* Technical report TR92-27. Ames, IA: Iowa State University of Science and Technology, Department of Computer Science, August.

———. 1993. "Analyzing software requirements errors in safety-critical, embedded systems." In *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on,* 126–133.

Lutz, Robyn R., and I Carmen Mikulski. 2003. "Operational anomalies as a cause of safety-critical requirements evolution." *Journal of Systems and Software.*

Lutz, Robyn R., and R M Woodhouse. 1996. "Contributions of SFMEA to requirements analysis." In *Requirements Engineering, 1996, Proceedings of the Second International Conference,* 44–51. IEEE Comput. Soc. Press.

Macaulay, L. 1993. “Requirements capture as a cooperative activity.” In *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on,* 174–181. IEEE Comput. Soc. Press.

Mäder, Patrick, and Jane Cleland-Huang. 2010. “A Visual Traceability Modeling Language.” In *Model Driven Engineering Languages and Systems,* edited by Dorina Petriu, Nicolas Rouquette, and Øystein Haugen, 226–240. Springer Berlin / Heidelberg.

Maiden, N A M. 1998. “CREWS-SAVRE: Scenarios for Acquiring and Validating Requirements.” *Automated Software Engineering* 5 (4): 419–446.

Maiden, Neil, and S Jones. 2004. *AN INTEGRATED USER-CENTRED REQUIREMENTS ENGINEERING PROCESS, Version 4.1.* City University London, April.

Martin, James, and James J Odell. 1998. *Object-oriented methods : a foundation.* Upper Saddle River, N.J.: Prentice Hall PTR.

Matulevičius, Raimundas, and Patrick Heymans. 2007. “Visually Effective Goal Models Using KAOS.” In *Advances in Conceptual Modeling – Foundations and Applications. ER 2007 Workshops CMLSA, FP-UML, ONISW, QoIS, RIGiM, SeCoGIS, Auckland, New Zealand, November 5-9, 2007. Proceedings,* edited by Jean-Luc Hainaut, Elke A Rundensteiner, Markus Kirchberg, Michela Bertolotto, Mathias Brochhausen, Yi-Ping Phoebe Chen, Samira Si-Saïd Cherfi, et al., 265–275. Berlin, Heidelberg: Springer Berlin Heidelberg.

Matulevičius, Raimundas, Nicolas Mayer, Haralambos Mouratidis, Eric Dubois, Patrick Heymans, and Nicolas Genon. 2008. “Adapting Secure Tropos for Security Risk Management in the Early Phases of Information Systems Development.” In *Advanced Information Systems Engineering,* 541–555. Berlin, Heidelberg: Springer Berlin Heidelberg.

Meier, Silvio, T Reinhard, R Stoiber, and Martin Glinz. 2007. “Modeling and Evolving Crosscutting Concerns in ADORA.” In *Aspect-Oriented Requirements Engineering and Architecture Design, 2007. Early Aspects at ICSE: Workshops in,* 3–3. May.

Mich, Luisa, Mariangela Franch, and Pierluigi Novi Inverardi. 2004. “Market research for requirements analysis using linguistic tools.” *Requirements Engineering* 9 (1): 40–56.

MIL-STD-882E. System Safety. 2012. Department of Defense of the United States of America, May.

Moody, Daniel Laurence. 2009. “The ”Physics” of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering.” *IEEE Transactions on Software Engineering* 35 (6): 756–779.

Moody, Daniel Laurence, Patrick Heymans, and Raimundas Matulevičius. 2009. “Improving the Effectiveness of Visual Representations in Requirements Engineering: An Evaluation of i* Visual Syntax.” In *Requirements Engineering Conference, 2009. RE ’09. 17th IEEE International,* 171–180. Los Alamitos, CA, USA: IEEE Computer Society, August.

Moody, Daniel, and Jos van Hillegersberg. 2009. “Evaluating the Visual Syntax of UML: An Analysis of the Cognitive Effectiveness of the UML Family of Diagrams.” In *Software Language Engineering. First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers,* edited by Daniel Laurence Moody, 16–34. Berlin, Heidelberg: Springer Berlin Heidelberg.

Moreira, Ana, Georg, Geri, and Mussbacher, Gunter, eds. 2013. “Third International Comparing *Requirements* Modeling Approaches (CMA@RE) workshop.” Accessed August 14, 2014. `http://cserg0.site.uottawa.ca/cma2013re/`.

Mullery, G.P. 1985. “Acquisition — environment.” In *Distributed Systems: Methods and Tools for Specification An Advanced Course,* 45–130. Berlin, Heidelberg: Springer Berlin Heidelberg.

Mylopoulos, John, Lawrence Chung, and B Nixon. 1992. “Representing and using nonfunctional requirements: a process-oriented approach.” *IEEE Transactions on Software Engineering* 18 (6): 483–497.

Ng, Annie W Y, Kin Wai Michael Siu, and Chetwyn C H Chan. 2012. “The effects of user factors and symbol referents on public symbol design using the stereotype production method.” *Applied Ergonomics* 43, no. 1 (January): 230–238.

OMG (Object Management Group). 2000. *OMG Unified Modeling Language Specification Version 1.3.* OMG, March.

———. 2008. *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Specification. Version 1.1,* May.

OMG (Object Management Group). 2011a. *Business Process Model and Notation (BPMN).* Object Management Group (OMG), January.

———. 2011b. *OMG Unified Modeling Language (OMG UML), Infrastructure Version 2.4.1,* August.

———. 2011c. *OMG Unified Modeling Language (OMG UML), Superstructure Version 2.4.1,* August.

———. 2012. *OMG Systems Modeling Language (OMG SysML)* Version 1.3. Object Management Group (OMG), June.

———. 2013a. *OMG Meta Object Facility (MOF) Core Specification* Version 2.4.1 (2013), June.

———. 2013b. *OMG Unified Modeling Language (OMG UML) Version 2.5 FTF Beta 2,* September.

———. 2014a. "UML." Accessed February 18. `http://www.omg.org/spec/UML/`.

———. 2014b. "XMI of the merged L3 UML 2.4.1 as an instance of UML 2.4.1 using XMI 2.4.1." Accessed April 12. `http://www.omg.org/spec/UML/20110701/UML.xmi`.

———. 2015a. *Interaction Flow Modeling Language,* March.

———. 2015b. *OMG Meta Object Facility (MOF) Core Specification Version 2.5,* June.

———. 2015c. *OMG Unified Modeling Language (OMG UML) Version 2.5,* June.

Office of Administrative Law, ed. 2014. "California Code of Regulations." Accessed July 21. `https://govt.westlaw.com/calregs/Index`.

Olivé, Antoni. 2007. *Conceptual Modeling of Information Systems.* Berlin, Heidelberg: Springer Berlin Heidelberg.

Omcan Inc. 2014. "SAUSAGE STUFFERS - Hydraulic Stuffers." Accessed August 6. `http://www.omcan.com/sausagestuffers/sausagestuffersdetail/hydraulicstuffers.html`.

Ott, Daniel. 2012. "Defects in natural language requirement specifications at Mercedes-Benz: An investigation using a combination of legacy data and expert opinion." In *Requirements Engineering Conference (RE), 2012 20th IEEE International.* IEEE Computer Society, September.

Ould, Martyn A. 1995. *Business processes : modelling and analysis for re-engineering and improvement.* Chichester; New York: Wiley.

Parnas, David Lorge. 1972. "A technique for software module specification with examples." *Communications of the ACM* 15, no. 5 (May): 330–336.

Parnas, David Lorge, and Jan Madey. 1995. "Functional documents for computer systems." *Science of computer programming* 25 (1): 41–61.

Pohl, Klaus, Günter Böckle, and Frank van der Linden. 2005. *Software Product Line Engineering - Foundations, Principles, and Techniques.* Springer.

Potts, Colin. 1991. "Seven (plus or minus two) challenges for requirements research." In *Software Specification and Design, 1991., Proceedings of the Sixth International Workshop on,* 256–259. IEEE Comput. Soc. Press.

———. 1995. "Using schematic scenarios to understand user needs." In *DIS '95: Proceedings of the 1st conference on Designing interactive systems: processes, practices, methods, & techniques.* ACM Request Permissions, August.

Reinhard, Tobias, Silvio Meier, Reinhard Stoiber, Christina Cramer, and Martin Glinz. 2008. "Tool support for the navigation in graphical models." In *the 13th international conference,* 823–826. New York, New York, USA: ACM Press.

Respect-IT sa. 2007. "A KAOS Tutorial." October. Accessed February 18, 2014. `http://www.objectiver.com/fileadmin/download/documents/KaosTutorial.pdf`.

———. 2014. "Objectiver: HomePage." Accessed February 18. `http://www.objectiver.com/index.php?id=4`.

Reubenstein, H. B., and R. C. Waters. 1989. "The requirements apprentice: an initial scenario." In *IWSSD '89: Proceedings of the 5th international workshop on Software specification and design.* ACM, April.

Reubenstein, Howard B. 1990. *Automated Acquisition of Evolving Informal Descriptions.* Technical report ADA225621. Fort Belvoir, USA: MIT Artificial Intelligence Laboratory, June.

Reubenstein, Howard B, and R. C. Waters. 1991. "The Requirements Apprentice: automated assistance for requirements acquisition." *IEEE Transactions on Software Engineering* 17, no. 3 (March): 226–240.

Robertson, Suzanne, and James Robertson. 2012. *Mastering the Requirements Process: Getting Requirements Right.* Addison-Wesley.

Rolland, C, C Souveyet, and C B Achour. 1998. "Guiding goal modeling using scenarios." *IEEE Transactions on Software Engineering* 24, no. 12 (December): 1055–1071.

Rolland, Colette, and Camille Ben Achour. 1998. "Guiding the construction of textual use case specifications." *Data & Knowledge Engineering,* no. 25: 125–160.

Rolland, Colette, and Camille Salinesi. 2009. "Supporting Requirements Elicitation through Goal/Scenario Coupling." In *Conceptual Modeling: Foundations and Applications,* 398–416. Berlin, Heidelberg: Springer Berlin Heidelberg.

Ross, D T, and K E Schoman Jr. 1977. "Structured Analysis for Requirements Definition." *IEEE Transactions on Software Engineering* 3 (1): 6–15.

Rumbaugh, James, Ivar Jacobson, and Grady Booch. 2004. *The Unified Modeling Language Reference Manual.* 2nd ed. Pearson Higher Education, July.

Schmid, Reto, Johannes Ryser, Stefan Berner, Martin Glinz, Ralf Reutemann, and Erwin Fahr. 2000. "A Survey of Simulation Tools for Requirements Engineering."

Schneider, Florian, Bernd Bruegge, and Brian Berenbach. 2013a. "A tool implementation of the unified requirements modeling language as enterprise architect add-in." In *Requirements Engineering Conference (RE), 2013 21st IEEE International,* 334–335.

———. 2013b. "The unified requirements modeling language: Shifting the focus to early requirements elicitation." In *Comparing Requirements Modeling Approaches Workshop (CMA@RE), 2013 International,* 31–36.

Schneider, Florian, Helmut Naughton, and Brian Berenbach. 2012. "A modeling language to support early lifecycle requirements modeling for systems engineering." *Procedia Computer Science* 8:201–206.

Selic, Bran. 2007. "A systematic approach to domain-specific language design using UML."

———. 2011. "The Theory and Practice of Modeling Language Design for Model-Based Software Engineering—A Personal Perspective." In *Generative and Transformational Techniques in Software Engineering III,* 290–321. Springer Berlin / Heidelberg.

Seybold, C, Silvio Meier, and Martin Glinz. 2004. "Evolution of requirements models by simulation." In *Principles of Software Evolution, Eighth International Workshop on,* 43–48. September.

Seybold, Christian, Silvio Meier, and Martin Glinz. 2006. "Scenario-driven modeling and validation of requirements models." In *Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools,* 83–89. New York, NY, USA: ACM.

Sharp, Helen, Anthony Finkelstein, and Galal Galal. 1999. "Stakeholder identification in the requirements engineering process." In *Database and Expert Systems Applications, 1999. Proceedings. Tenth International Workshop on,* 387–391.

Sikora, Ernst, Bastian Tenbergen, and Klaus Pohl. 2012. "Industry needs and research directions in requirements engineering for embedded systems." *Requirements Engineering* 17 (1): 57–78.

Simos, M, D Creps, C Klingler, L Levine, and D Allemang. 1996. *Organization Domain Modeling (ODM) GuideBOOK, Version 2.0.* Technical report STARS–VC–A025/001/00. Manassas, VA: Lockheed Martin Tactical Defence Systems.

Sindre, G, and Andreas L Opdahl. 2000. "Eliciting security requirements by misuse cases." In *Technology of Object-Oriented Languages and Systems, 2000. TOOLS-Pacific 2000. Proceedings. 37th International Conference on IS -,* 120–131. IEEE Comput. Soc.

Sirman SpA. 2014. "Product List / Sausage Stuffers." Accessed August 6. `http://www.sirman.com/en/product-list/sausage-stuffers.html`.

Sousa, Kenia, Jean Vanderdonckt, Brian Henderson-Sellers, and Cesar Gonzalez-Perez. 2012. "Evaluating a graphical notation for modelling software development methodologies." *Journal of Visual Languages and Computing* 23, no. 4 (August): 195–212.

Sparx Systems Pty Ltd. 2014. "Enterprise Architect Product Website." Accessed March 27. `http://www.sparxsystems.com/products/ea/index.html`.

Spivey, J M. 1998. *The Z Notation: A Reference Manual.* 2nd ed. Programming Research Group University of Oxford: Prentice Hall.

Stoiber, Reinhard, Silvio Meier, and Martin Glinz. 2007. "Visualizing Product Line Domain Variability by Aspect-Oriented Modeling." *Requirements Engineering Visualization, 2007. REV 2007. Second International Workshop on:* 1–6.

Sutcliffe, A, and Pete Sawyer. 2013. "Requirements elicitation: Towards the unknown unknowns." In *Requirements Engineering Conference (RE), 2013 21st IEEE International,* 92–104. July.

Talsa. 2006. *H15 * H20 * H26 * H31 * H42 * H52.* 3.5. October.

Teichroew, Daniel. 1972a. "A survey of languages for stating requirements for computer-based information systems." In *the December 5-7, 1972, fall joint computer conference, part II,* 1203–1224. New York, New York, USA: ACM Press.

———. 1972b. "Computer-aided documentation of user requirements." In *INFORMATION SYSTEMS FOR MANAGEMENT,* edited by Fred Gruenberger, 97–112. Englewood Cliffs, NJ: Prentice Hall.

Terzieva, Kameliya. 2011. "Stakeholder Specific Viewing, Filtering, and Workflow Support in URML." Master's thesis, Technische Universität München.

United States Department of Labor - Occupational Safety and Health Administration (OSHA). 2011. "Accident: 201128717 - Employee Amputates Finger Cleaning A Sausage Machine." December. Accessed February 18, 2014. `https://www.osha.gov/pls/imis/accidentsearch.accident_detail?id=201128717`.

"User Stories." 2016. Accessed June 9, 2016. `https://www.agilealliance.org/glossary/user-stories/`.

Wobbrock, Jacob O, Htet Htet Aung, Brandon Rothrock, and Brad A Myers. 2005. "Maximizing the guessability of symbolic input." *CHI '05 Extended Abstracts on Human Factors in Computing Systems:* 1869–1872.

Wüest, Dustin, and Martin Glinz. 2011. "Flexible sketch-based requirements modeling." In *Requirements Engineering: Foundation for Software . . .*

Xia, Yong. 2004. "A Language Definition Method for Visual Specification Languages." PhD diss., Universität Zürich.

Yeh, R T, and Pamela Zave. 1980. "Specifying software requirements." *Proceedings of the IEEE* 68 (9): 1077–1085.

Yourdon, Edward. 1988. *Modern Structured Analysis.* 1st international edition. Prentice Hall, December.

Yu, Eric. 1993. "Modeling organizations for information systems requirements engineering." In *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on,* 34–41.

———. 1997. "Towards modelling and reasoning support for early-phase requirements engineering." In *Requirements Engineering, Proceedings of the Third IEEE International Symposium on,* 226–235.

Yue, Kaizhi. 1986. "Constructing and analyzing specifications of real world systems." PhD diss., Stanford Univ., CA (USA).

———. 1987. "What does it mean to say that a specification is complete?" In *4th International Workshop on Software Specification and Design,* 42–49. April.

Zave, Pamela. 1979. "A comprehensive approach to requirements problems." In *Computer Software and Applications Conference, 1979. Proceedings. COMPSAC 79. The IEEE Computer Society's Third International,* 117–122. IEEE.

———. 1980. *The Operational Approach to Requirements Specification for Embedded Systems.* Technical report ADA097273. Defense Technical Information Center (DTIC), December.

Zave, Pamela, and Raymond T Yeh. 1981. *Executable requirements for embedded systems.* IEEE Press, March.