Institut für Informatik
der Technischen Universität München

Specification and analysis of adaptive systems

Bernd Spanfelner

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. Dr. Arndt Bode

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Dr. h.c. Manfred Broy
2. Univ.-Prof. Dr. Wolfgang Reif

Die Dissertation wurde am 22.09.2014 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 31.03.2015 angenommen.

Acknowledgements

My thanks go to Prof. Dr. Dr. h.c. Manfred Broy for enabling me to work at the chair for Software & Systems Engineering, participating in many interesting research projects, and finally giving me the opportunity to write this thesis. The fruitful atmosphere, broad range of topics and many opportunities for discussions have always been an inspiration and finally resulted in some ideas that found their way into the thesis. His comments and sometimes challenging questions

Additional thanks go to Prof. Dr. Wolfgang Reif for being my second PhD thesis supervisor and supporting my efforts.

Further thanks go to Sabine Rittmann, Doris Wild, Christian Leuxner, Wassiou Sitou, Michael Fahrmair, Alik Harhurin and Bernhard Schätz for supporting me in different stages of my work, listening to me and being often available for discussions. Special thanks go to Sabine Rittmann, Wassiou Situo, and Alik Harhurin for reading drafts of this thesis and providing feedback.

I am also glad to had the opportunity to work with a number of skilled researchers of which I want to mention Daniel Mendez Fernandez, Stephan Kugele, Stephano Merenda, and many more.

Especially I want to thank Stephano Merenda for having a similar mind. The Fridays eating "Weißwürste" were a pleasure and a delight in some stressful situations.

This PhD thesis took a couple of years that brought many changes. Without the support of my family and friends it would not have been possible to complete the thesis. Thank you, Christian and David for easing my mind with various leisure activities. Furthermore, I want to thank my Mum and Dad for supporting me and encouraging me to go my own way.

Finally I want to thank my wife Stephanie for being there and believing in me even if I found "yet another problem" in the thesis. You were patient and supported me even after this long time. And I want to thank my daughter Lilja Rosalie and my son Julian Raphael for being such an enrichment and bringing even more luck to my life.

Abstract

Software systems that are capable of automatically adapting the offered functions to varying application scenarios impose additional complexity to their development. Adaptations require considering the proximate environment, its evaluation and interpretation, and finally a decision about offered functions. This requires additional functions that need careful treatment. Insufficient engineering easily results in inappropriate or even harmful system behaviors. If the adaptations miss to meet the expectations of their users, the system acceptance decreases.

This thesis presents the MARLIN specification technique for specifying and analyzing context-adaptive systems consisting of the specification language and an embedding in common methodologies.

The MARLIN language models interactions with the system as services. The services can be combined using different composition operators. This allows building more complex services from simpler ones. A special operator puts the services into a defined context according to the situations they shall be available in.

While the language provides the tools to model adaptive behavior the MARLIN methodology embeds the development of context-adaptive systems into the early phases of system development. The methodology considers the different application scenarios and systematically transfers the according requirements into a formal system specification.

Initially requirements are often inconsistent and incomplete. Furthermore, requirements that are valid for different situations must be related. Therefore, the MARLIN methodology includes an analysis of specifications. The focus is on detecting and removing deficiencies. This includes treating the defined situations and their relations. Therefore, MARLIN bases on a formal approach and has a formal semantics that allows using algebraic laws for restructuring the specification to separate the behaviors according to the defined application scenarios. The separated behaviors are independent and formal proof systems can analyze them apart. The result is a consistent and – with respect to the function domain – complete specification that serves as the basis for any subsequent development activities.

Contents

Chapter 1

Introduction

This thesis presents the MARLIN specification technique for embedded and context-adaptive systems. MARLIN formalizes requirements as functions called services and combines them with respect to different situations where they shall be effective. The formalization enables an analysis for certain deficiencies and their discussion with the respective stakeholders. As a result MARLIN supports creating a consistent specification of the system which serves as a starting point for a refinement-based development approach. In this chapter we introduce the reader to the topic and the contributions of the thesis.

Contents

1.1. Motivation for context-adaptive systems

A growing number of applications along with the fast development of new technologies like smaller and more powerful processors, sensors, and actuators lead to a pervasion of our everyday life with computer systems. Especially, embedded systems support us in complex or even dangerous tasks. Starting with a famous article of Mark

Weiser [Weiser, 1991] describing computing in the 21st century the idea of *ubiquitous computing* gained more and more attention. His vision was the deployment of small devices virtually everywhere to support the users. More and more functions on these devices collaborate to provide even more sophisticated functions.

As a serious accompanying phenomenon users need to interact with the systems quite often to control their behavior or to trigger their collaboration. The complexity and number of offered functions excessively attracts the attention of users resulting in an unwanted distraction from their actual tasks. To relieve users from distractions the vision of Weiser also includes the removal of conscious interactions with computers. The systems have to "guess" the expected behavior to avoid direct operation.

Context-adaptive systems use additional information to relieve users from as many interactions as possible [Schilit et al., 1994; Geihs, 2008; Oreizy et al., 1999; Coutaz et al., 2005; Aksit and Choukair, 2003; Baldauf et al., 2007]. Informally spoken, context-adaptive systems observe the users' context and adapt their offered functions and their behavior according to identified situations [Biegel and Cahill, 2004]. This enables their use even if the user has only limited interaction capabilities (e.g., while driving a car). Context adaptation considerably eases the usage and hence increases the usability of system.

1.2. Problem statement

It is challenging to develop context-adaptive systems. Most significantly, automatic adaptations have to meet the expectations of stakeholders and need to be appropriate for recognized situations. If they fail to meet the expectations of stakeholders, their acceptance decreases. Inappropriate behavior of mission critical[1] or even safety relevant[2] systems may cause additional costs or even harms to people.

The development of context-adaptive systems has different challenges throughout the development process. Subsequently we introduce an idealized software development approach to motivate and explain the particular challenge that we address in this thesis.

1.2.1. An idealized development approach

In Figure 1.1 we pick up a picture of Broy [Broy, 2007] to explain the role and use of formal specifications and deduce the challenges we face in this thesis. The figure shows a coarse outline of an idealized development process.

After creating the functional specification as a formalization of the requirements the engineers elaborate an architecture specification as a refinement of the functional specification by iteratively refining components at one level of abstraction into the next

[1] Systems whose failure produces high costs or heavily affects business goals

[2] Systems whose failure causes harm to users like, e.g., those with a high (A)SIL-level as introduced in the standards IEC61508 and ISO26262

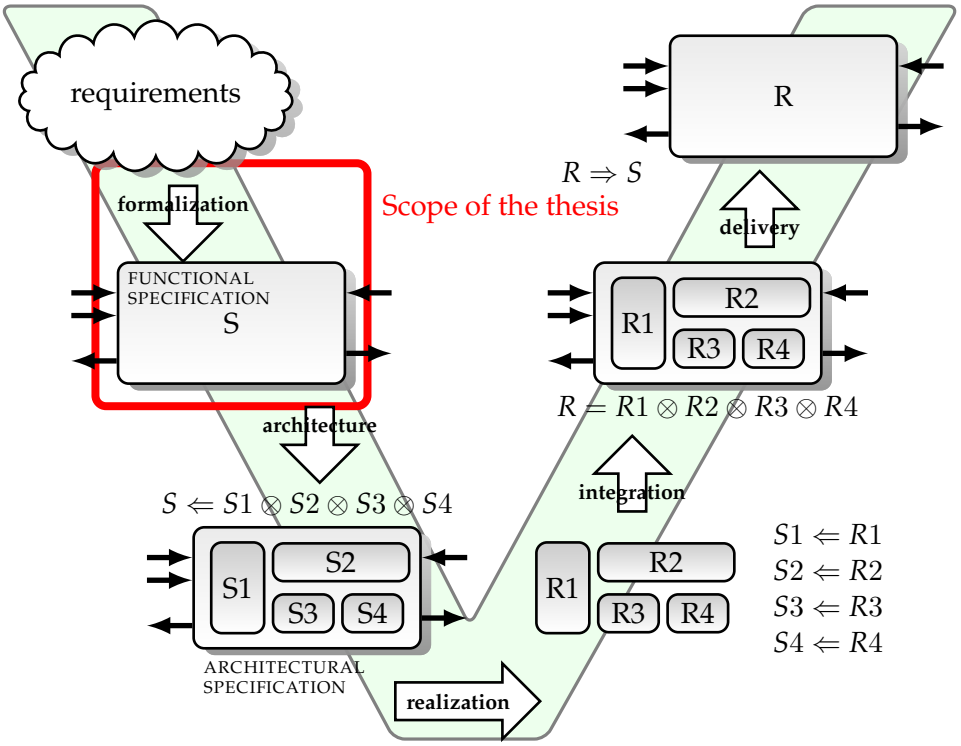


Figure 1.1.: Outline of an idealized development process according to Broy. The artifacts are organized according to their occurrence in the V-Modell [Rausch and Niebuhr, 2005].

one. Formally, the refinement corresponds to a logical implication. Verification demonstrates that the generated artifacts satisfy the specified characteristics.

The implementation of leave level components follows the specification of the architecture. This comprises design decisions about algorithms, etc. By module verification one shows that each module implementation satisfies its specification given in the architecture. Again, this is a refinement relation, and formally denoted as an implication.

In this chain of implications artifacts are specifications for the next lower level of abstraction and take the role of proof obligations. The rigorous application of this approach ensures the correctness of the implementation with respect to the specification. The architecture guides the iterated integration of modules and components into a realization which is provable correct with respect to the functional specification by the chain of implications.

1.2.2. The need for qualified specifications

The idealized development approach shows that the basis of a successful software development process is a high quality functional specification. Specifications describe all acceptable implementations [Heitmeyer et al., 1996]. Well-formed specifications lead to systems with an acceptance of a broad range of stakeholders because they satisfy the

stakeholders' requirements.

Poor specifications are a root cause of failing system development projects. Heitmeyer et al. identify deficiencies and errors in the requirements and, hence, in the functional specifications as large cost drivers for system development projects [Heitmeyer et al., 1996]. The chain of implications in the idealized development approach emphasizes this: any inappropriate requirement in the functional specification will be provable realized in the implementation. If requirements in the specification are conflicting, no implementation exists at all that satisfies the requirements.

The growing complexity of systems is one cause for deficiencies of specifications. Context-adaptive systems are particularly complex due to their multi-functional nature accompanied by a complex rule set for the systems' adaptations: Biegel et. al. identify the complexity of specifying effective rules for the adaptation of applications as one of the biggest challenges for context-adaptive systems [Biegel and Cahill, 2004].

According to Harhurin [Harhurin, 2010], different stakeholders describe systems according to their different viewpoints. A functional specification integrates these viewpoints. An ad hoc integration of the viewpoints of stakeholders into a specification bears the risk of distorting the requirements if the viewpoints initially are contradicting, ambiguous, or incomplete.

To address these issues, integrating such viewpoints requires a restructuring or even reformulation of the requirements possibly resulting in an insufficient or even incorrect integration of requirements. Possible different views in different situations aggravate this situation for context-adaptive systems.

A specification failing to reflect the actual requirements of stakeholders or describing them ambiguously leads to a final product with unintended behaviors. On the one hand, this leads to unexpected behavior (UB) [Fahrmair et al., 2006b] during operation, and, on the other hand, makes it difficult to assign responsibilities.

It is a better approach to analyze the requirements for the listed deficiencies and to discuss them with the stakeholders to make agreed changes. A proper structure of the specification that reflects the viewpoints of stakeholders and the considered situations supports their validation.

All three aspects together, the complexity of multi-functional systems, the situation dependent availability of functions, and the different views of stakeholders, raise the need for an appropriate specification technique.

1.2.3. The need for consistent specifications

If requirements are incomplete and inconsistent at the beginning, a specification technique should support their analysis to reveal these deficiencies. The verification steps outlined in Figure 1.1 only work, if the functional specification is consistent. This becomes clear if we think of specifications as collections of logical formulas characterizing the system behavior. These formulas are the axioms that any proper implementation must satisfy. Concerning mathematical logic, contradictions prevent the existence of models that satisfy the axioms. As a consequence the specification is not satisfiable.

According to the idealized development approach no application specific proof obligation exists for a functional specification. Therefore, we use the properties of consistency and input completeness as general purpose properties that functional specification must satisfy. Note, there is no means to ensure the completeness of requirements in general, i.e., to ensure that the specification contains all requirements the system shall meet [IEEE, 1998]. This requires validation of the requirements with the stakeholders and is not possible by mathematical verification. However, the notion of input completeness reveals yet unconsidered scenarios in a specification, if it misses proper system reactions for some inputs.

A proper analysis of context-adaptive system specifications shall take into account that usage functions (and hence the related requirements) are situation dependent. As a consequence, the analysis must take care of their peculiarities:

- The situations where requirements apply may be disjoint, overlapping, or congruent. Therefore, the consistency of the requirements depends on the relation of the respective situations they are effective in. Hence, we need to find and analyze every occurring combination of usage functions.

- There may be situations with no defined usage functions. We need to identify and discuss them with the stake holders to avoid unexpected behavior.

1.2.4. How to address the challenges

The work at hand addresses the functional specification of context-adaptive systems. Figure 1.1 presents the scope as a red rectangular. The identified challenges for the specification of context-adaptive systems are:

Means for managing complexity A separate modeling of requirements according to different viewpoints of stake holders is necessary. Proper compositions allow their integration without the need of changing them to be consistent yet. Analysis methods enable revealing inconsistencies and contradictions to address them later in agreement with the stakeholders.

Explicit model of the adaptation logic The adaptation logic must be identifiable to support its validation and to allow a straight forward mapping of situational requirements into the formal framework. In addition to the different views of stake holders, the different situations need to be described apart and properly integrated. The analysis of specifications must take into account the varying effectiveness of usage functions.

Appropriate level of abstraction A specification describes the problem space and avoids a premature restriction of the solution. Therefore, a specification neglects structural aspects of the system and regards inputs as abstract events instead of concrete signals.

1.3. State of the art

In summary, no formalism we are aware of is appropriate to specify the functionality of context-adaptive systems in the very early phases. Some are inappropriate to model possibly conflicting behaviors in the early phase of development allowing their analysis while abstracting from structure and distribution. Others lack appropriate means to capture the relevant aspects of context-adaptive systems or only provide rudimentary support.

We will give detailed information about related work in the respective chapters. However, we anticipate a short overview over the state of the art in specifying context-adaptive systems to support our claim. We organize the overview into context-adaptive-system-specific approaches and general-purpose formalisms.

context-adaptive systems A working group around Bill Schilit developed the first context-adaptive systems. They described new applications, developed the vision and explored the technical possibilities rather than considering a systematic approach of context-adaptive-system development. However, even at this time, people already noticed that context is more than location and that a decent consideration of the context and its impact on the system behavior is a prerequisite to a successful application design [Schilit et al., 1994; Schmidt et al., 1999].

Kolos et al. present some related work in the field of requirements engineering methods for context-adaptive systems [Kolos-Mazuryk et al., 2006]. Goal-oriented approaches like, e.g., KAOS [Darimont et al., 1997] and I^* [Yu, 1997], started to include the users current goals into the considerations although a systematic and explicit treatment of context is still missing.

Personal and Contextual Requirements Engineering (PC-RE) [Sutcliffe et al., 2006] and the ReCawar approach [Sitou and Spanfelner, 2007; Sitou, 2009] are scenario-based approaches. The developers derive the requirements from a set of application scenarios which are explicitly enriched by context information. The ReCawar approach, e.g., provides a meta model of context and describes the systematic construction of a context model for the application under development. Instances of that model (called situations) relate to application requirements. Both approaches focus on the elicitation of requirements and the situations of their effectiveness. With respect to the development process outlined in Figure 1.1, they lack a technique for transferring the requirements into a formal specification or to analyze them for inconsistencies in a rigorous way.

Other approaches engage in architectures and tool-kits that manage the access to context information for a broad range of applications [Dey et al., 2001; Schilit, 1995]. Fahrmair proposes an integrated model of context and applications [Fahrmair, 2005] that offers dynamically changing architectures to overcome *Unexpected Behavior* (UB) [Fahrmair et al., 2006b] or *Automation Surprise* [Sarter et al., 1997]. A number of similar approaches are, e.g., discussed in [Baldauf et al., 2007]. These approaches focus on defining architecture specifications instead of functional specifications of systems. Furthermore, some formalisms only provide an implicit semantics given by an imple-

mentation of the framework or are informal. They lack (explicit) means for an analysis.

Trapp presents an approach that focuses on the formal specification of system behavior presented as (synchronously) communicating components [Trapp, 2005]. Schäfer enhances this approach by describing the analysis of the resulting specifications [Schaefer, 2008]. Again, the use of components defines an architecture for the system and therefore is more useful during the architecture design. The analysis methods of Schäfer pick up the component structure and verify them against a set of application specific properties. This corresponds to the architecture verification shown in Figure 1.1. We regard these properties as part of the functional specifications and are interested in a consistent provision of the like.

General formalisms In literature, a number of formalisms exist that are appropriate for describing general system behavior. Prominent examples are process algebras like CSP [Hoare, 1978] and CCS [Milner, 1982] and their virtually countless derivatives and extensions. FOCUS[Broy and Stølen, 2001], Lustre [Caspi et al., 1987] and Esterel [Berry and Gonthier, 1992] are representatives that include a data-flow view. I/O-Automata [Lynch and Tuttle, 1989] are another popular formalism for the description of behavior. While the different semantics of these approaches are of general use, their composition operators focus on modularity and parallelism of components. They facilitate information hiding for components and impose structural restrictions that prevent conflicts (i.e., it is not possible to model conflicts). They are more useful for defining the logical architecture than for the black-box view including overlapping (initially inconsistent) viewpoints as required for the functional specification.

Approaches like Software Cost Reduction (SCR)[Heitmeyer et al., 1996] and state charts [Harel, 1987] are more suitable for functional specifications and even provide concepts of modes and hierarchical states respectively. However, apart from the known issues with state charts [von der Beeck, 1994], the mode concepts (or similar the hierarchic states) are inadequate for capturing the situation dependent aspects of context-adaptive systems. Situations are phases of execution during which a certain behavior is observable. A well-suited formalism for specifying context-adaptive systems enables characterizing these phases apart from the definition of the functions that are active in each of the situations. SCR and state charts especially support no separate description of the functions offered by the system and the logic that chooses the functions according to an identified situation.

1.4. Content of the thesis

We summarize the contributions of this thesis with respect to the identified challenges. Furthermore, we classify the approach with respect to a model of levels of abstraction for the development of embedded systems and the approached system class.

1.4.1. Contribution

The work at hand presents MARLIN, an approach for generating consistent functional specifications out of informal (and potentially conflicting) requirements for context-adaptive systems.

The approach is a scenario-based approach. It transfers use-cases into formal specifications of system behavior. In contrast to the scenario-based approaches like Sutcliffe and Sitou [Sutcliffe et al., 2006; Sitou, 2009] MARLIN offers an analysis of system specifications rather than the elicitation of requirements. MARLIN is a subsequent step of these approaches and has three major aspects.

A formal definition of context-adaptive systems and related terms like situation and context-model.

A formal language called the MARLIN language for specifying the behavior of systems in different contexts and from different views of stake-holders.

Methodological guidance to the application of MARLIN including the analysis of specifications for inconsistencies and partiality.

We call all three aspects together the MARLIN specification technique.

1.4.1.1. Scope of the approach

In addition to the focus on consistent and input complete functional specifications that serve as proof obligations for subsequent development steps, the approach fits best for a certain class of systems:

Mission-critical systems By mission critical we refer to systems whose application domain justifies additional efforts for ensuring correctness. It should be clear that applying formal methods and verification techniques causes additional costs. Mission critical systems can cause harm (additional costs, loss of prestige or even harm to people) that exceeds the additional costs for applying the method. Avionics and automotive are typical application domains, especially if the systems have an elevated (A)SIL level. The actual additional costs of applying MARLIN, of course, depend on the application. Domain experts should carefully assess a decision in favor of MARLIN.

Embedded systems The approach aims at modeling embedded systems. These systems control physical processes and therefore often are mission critical. Embedded systems combine hard- and software. This integration needs considerations at an abstract, functional level. Furthermore, these abstractions often create a manageable state space. Restrictions of formal proof systems limit the analysis of the specifications with large state spaces. Many (tool supported) proof systems require finite or at least countable infinite many states to succeed.

Time-discrete systems MARLIN uses a system model of discrete time. Therefore, it is best used with systems that can be specified by reactions to discrete events. Hybrid or continuous systems are out of scope.

Context-adaptive systems The approach focuses on context-adaptive systems, especially on the specification of the adaptation logic. The basic behaviors that are chosen by an adaptation logic are conventional behaviors. Therefore, the approach includes means to specify all kinds of behaviors. The focus for the adaptation logic within this thesis is furthermore on a certain sub set of context-adaptive systems:

- Parametric and functional adaptations whose functions can be separated into discrete classes. The classes represent behaviors that are valid in certain discrete contexts.
- Predetermined adaptations that need to be analyzed to guarantee certain properties. Designers decide the relation between context and provided function at design time. We exclude any runtime changes to the adaptation logic.

1.4.1.2. Formal definition of terms

The definition of context-adaptive systems seems to be intuitively clear, but when it comes to a precise definition, the existing definitions leave room for interpretations. Instead of giving another textual definition based on the usage of context information as a discriminating factor, we give a *formal definition*. The formal definition covers the considered aspects and is even appropriate for topics that are out of the scope of the thesis (like sensors, data acquisition and related topics). The definitions cover the terms *context*, *situation*, and *context model* as well.

We classify context-adaptive systems with respect to three dimensions. The *user dimension* classifies the extent of adaptations from a user's point of view. The *system dimension* classifies the structure of context-adaptive systems and the use of context information. Finally, the *flexibility dimension* classifies rule sets for context-adaptive systems with respect to their flexibility at run-time.

1.4.1.3. Formal language

This thesis presents the MARLIN language, a formal specification language supporting the mapping of requirements given as, e.g., use-cases into a formal functional specification. MARLIN provides I) a modular specification of the usage functions of a system, II) the integration of different viewpoints on the usage functions, and III) the embedding into a logic that controls the activation of the usage functions.

The MARLIN language is a derivative of service-based modeling techniques [Schaetz, 2002; Broy, 2005; Broy et al., 2007] that build specifications from services, which represent aspects of the observable behavior and allow the description of overlapping, possibly conflicting behaviors. Such specifications formalize the functional requirements only.

The composition operators of MARLIN support the systematic use of structure and modularity while specifying context-adaptive systems. This eases the modeling of correlations between offered functions and situations together with the validation of their

appropriateness. The developers are able to identify and allocate divergences between the actual requirements of the stakeholders on the one side and the specification on the other side. The situation-centric specification imposes a structure that aids the management of complexity by separating the aspects of the usage functions from the actual logic to choose them. Especially, the composition operators allow a restructuring of the system that separates context-related aspects from interactions between users and system and matches to our definition of context-adaptive systems.

The MARLIN language has a formal semantics allowing for unambiguous specification and sophisticated analysis. A big-step semantics [Leroy, 2010], also called natural semantics [Kahn, 1988], provides an abstract characterization of systems' behaviors in a denotational style.

For mission critical systems, designers prefer guarantying certain properties over providing a maximum of flexibility [Trapp, 2005]. Hence, we explicitly exclude any of those aspects of context-adaptive systems that relate to learning and dynamic architectures and focus on a priori fixed behaviors.

1.4.1.4. Methodological guidance

We discuss aspects of applying MARLIN to the creation of functional specifications. We support the combination of MARLIN with service-based methodologies, e.g., described by Rittman [Rittmann, 2008].

We start with the assumption that an initial elicitation of requirements is finished and a list of elaborated requirements is already available. We do not assume that they are complete or consistent in any sense. We assume that those requirements are available as a set of use-cases and require a data model that describes the proximate environment at an adequate level of abstraction. The environment model is the basis for the definition of the different situations of use for the system. Our assumptions match the outcomes of the *ReCawar* methodology [Sitou and Spanfelner, 2007; Sitou, 2009] but may apply to any appropriate representation of a requirements description.

The MARLIN approach presents a procedure to transfer the available requirements and their contexts directly into a formal specification. This includes the handling of mutual dependencies of overlapping services (feature interactions [Zave, 1993]). We proof that feature interactions and context adaptation are closely related and show how MARLIN models both. A procedure called unbundling integrates services that are effective in different, possibly overlapping situations.

Figure 1.1 indicates that the decomposition of systems into sub-systems and the mapping of requirements to these sub-systems are recurring steps while creating a system architecture. We show how MARLIN integrates into this development of the architecture by applying MARLIN during the architectural work to specify sub-system behaviors as black boxes. Cause-effect-chains relate these steps.

The analysis of the formal specification is a major concern of the MARLIN specification technique. MARLIN exhibits a number of algebraic properties that are useful for restructuring the specification and for establishing certain normal forms. This supports the analysis of specifications by dividing the services into coherent subsets with the

option to analyze them apart. A small-step semantics similar to [Henzinger, 2000] and [Schätz, 2009] is input for the actual analysis. Conflicts in the specification are tracked to a limited subset of the system's requirements. MARLIN provides means to resolve conflicts without an extensive change of the specification. Hence, problem-identification and problem-handling are local.

1.4.2. Outline

The thesis has seven chapters and contains additional information about the case study and the algebraic properties in the appendix.

The basis for the approach is the FOCUS theory for concurrent systems [Broy and Stølen, 2001]. In Chapter 2, we give an introduction to the foundations of FOCUS together with its extension to services called JANUS [Broy, 2005]. The chapter presents the basic ideas of both together with the changes required for the MARLIN approach.

We define the most important terms related to context-adaptive systems on basis of a system model in Chapter 3. The chapter addresses the terms *context-adaptive system*, *context*, *context model* and *situation* together with classifications in the dimensions *user*, *system* and *flexibility*.

The introduction of the formal language MARLIN is the topic of Chapter 4 which starts with the explanation of the meaning of services for specifications. The chapter defines atomic services and their semantics. Simple composition operators cover aspects of composing behaviors and integrating views. Conditional composition restricts behaviors to phases of execution (situations). We discuss the peculiarities of conditional composition – especially of interruption and resumption of behaviour.

Chapter 5 addresses the methodological aspects of using MARLIN. Based on extensions of the original contextual requirements chunks of Sitou [Sitou, 2009], we describe the elaboration of a service hierarchy. We systematically treat already known conflicts and transfer context-related aspects into mode transition systems.

The analysis of specifications in chapter 6 describes the use and application of transformations with the goal to reveal yet undiscovered deficiencies. For this we establish a normal form that separates the usage functions and the adaptation logic and provide separate approaches for analyzing their specific deficiencies.

Finally, the conclusion and outlook in Chapter 7 summarize the approach and discusses the results.

Appendix A presents the algebraic aspects of MARLIN together with important properties and transformations for specifications that preserve their behaviors. Appendix B presents details of the case study that we sketch subsequently and use as a running example.

1.5. Case study

Throughout this thesis, we use a running example: a car access system. Appendix B presents details of the case study. After a brief overview over the most significant

aspects we refer to the case study throughout the thesis.

The original case study covers collaborative work with a German car vendor. We modified it in relevant technical details for this thesis to circumvent any issues with their intellectual property rights. The case study only contains generalizations and commonly used functionality.

In the case study, we regard a modern car equipped with a wireless key detection enabling functionality to access the car. To keep it simple, we only regard a small set of contextual aspects.

Figure 1.2 shows the context model (cf. Section 3.1.2) of the considered system. Sitou [Sitou, 2009] and Henricksen [Henricksen et al., 2005], e.g., explain the elaboration of the context model. An explicit context model allows the analysis of dependencies between a context-adaptive system and its environment and the definition of the situations the system has to be sensitive to. Alike, models enable the investigation of the suitability of available information for distinguishing the relevant situations. However, this kind of analysis is part of the requirements elicitation and out of scope of this thesis.

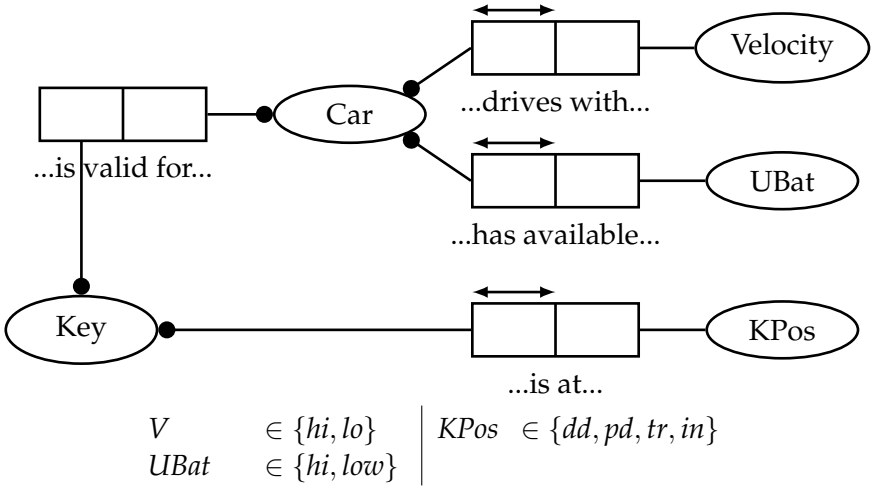


Figure 1.2.: Context model for the case study

The context model is in ORM (Object Role Model) notation as suggested by [Henricksen et al., 2005] and [Sitou, 2009]. We omit details of the used ORM notation and its advantages and refer to [Halpin, 1996] for an introduction. For the reader it is sufficient to read the diagram similar to the more common entity-relationship models [Chen, 1976]. In fact ORM is similar but provides some additional means to detail out the relations between entities (in ORM called Objects).

The objects `Key` and `Car` are only auxiliary and used to connect the other objects. They do not directly identify situations. However, they support the identification of objects that are actually relevant for the identification of situations during the requirements elicitation process. `KPos` is the information about the position of a valid key if the system detects one: (`dd` = driver door, `pd` = passenger door, `tr` = trunk, `in` = inside). In this case study the velocity `V` is either greater than zero (`hi`) or equal to zero (`lo`). This is sufficient for our purposes. Likewise, the battery voltage `UBat` is either high (`hi`) or

low (`lo`). In this case study we do not distinguish between different keys. This is useful only if, e.g., a certain key relates to user preferences.

Table 1.1 presents a clipping from the contextual requirements chunks that relate the system's functional requirements with their corresponding situation they are effective in. We present a more detailed list of the initial requirements in the appendix.

Nr.	Context	Requirement
	$V = hi$	Doors become locked, Doors cannot be unlocked or opened from outside
	$V = lo \ \wedge \ UBat = hi \ \wedge \ KPos = tr$	Trunk is opened on Gesture (with respect to. disturbance Protection)
	$V = lo \ \wedge \ UBat = hi$	Trunk is closed on gesture
	$V = hi \ \wedge \ KPos = tr$	Trunk is opened on touch
...		

Table 1.1.: A clipping from the requirements in CRC-style

In addition to the contextual inputs we define a set of user inputs:

$$\begin{array}{ll|ll} TA & \in \{t, op, cl\} & DL & \in \{l\} \\ DA & \in \{t, in\} & ER & \in \{on, off\} \\ PA & \in \{t, in\} & & \end{array}$$

The channel `ER` communicates a user's wish to start (`on`) or stop (`off`) the engine. Channels `DA`, `PA` and `TA` (driver-, passenger- and trunk door activation) carry information for the driver door, passenger door and trunk door respectively: the signal `t` indicates the activation of the respective outside door trigger, `in` indicates the activation of the in-car door trigger, `op` indicates the recognition of an opening gesture and `cl` the recognition of a closing gesture. The channel `DL` carries information about a user's wish to lock all doors.

The output interface:

$$\begin{array}{ll|ll} EC & \in \{on, off\} & TC & \in \{l, u\} \\ DC & \in \{l, u\} & TD & \in \{op, cl\} \\ PC & \in \{l, u\} & & \end{array}$$

The channel `EC` commands the engine to start (`on`) or halt (`off`). Channels `DC`, `PC` and `TC` communicate the driver door, passenger door and trunk door commands with `l` = lock and `u` = unlock. At channel `TD`, the signal `op` causes the trunk door engine to physically open the trunk door and the signal `cl` to physically close the door.

The original case study was modeled in the tool COLA (COmponent LAnguage)

[Kugele et al., 2007]. COLA is a CASE-Tool (Computer Aided Software Engineering Tool) that realizes a seamless development across the development. It provides means to model the functional specification, the logical, and the technical architecture [Broy et al., 2008]. COLA uses the approach of Rittmann [Rittmann, 2008] for the functional specification.

From the viewpoint of MARLIN it has certain restrictions. An alternative composition is missing to overcome conflicts and it lacks shared variables to model common aspects of functions. The application of modes is different from MARLIN's mode transition systems and less flexible. Therefore, workarounds implement, e.g., mode transition systems and alternative composition where necessary. These workarounds introduced a large number of internal communications and additional components. The lack of algebraic properties of COLA allows no application of the analysis methods described for MARLIN.

Chapter 2

Foundations

In this chapter we introduce a general purpose theory that serves as a mathematical model of system behavior. We base notions like *function*, *service*, and *system* on two theories: the FOCUS theory [Broy and Stølen, 2001] for the specification of reactive systems and its extension, the JANUS approach [Broy, 2005] that presents a generalization for services.

In Section 2.1 we take a look at messages and streams of messages. Subsequently, in Section 2.2 we define a couple of functions on streams that we use in definitions including the semantics of the MARLIN language in Chapter 4. In Section 2.3 we present means to characterize sets of streams that we will use in some specifications. Finally, we define the mathematical interpretation of a system behavior in Section 2.4.

Contents

2.1. Streams

To prepare the relevant definitions concerned with systems and services we start with some basic concepts of the FOCUS theory. The most basic concept is that of a type.

DEFINITION 1 (TYPE):
A type is a name for a carrier set. We use types to classify data. Let NAME be the universe of names and $\mathbb{D}$ be the universe of data. We define a function

$$TYPE : NAME \to \mathcal{P}(\mathbb{D});\ x \mapsto y$$

that returns the set of data y that belongs to the type with name x. With $T \in \mathrm{NAME}$ we define:

$$i \in T \stackrel{def}{=} i \in TYPE(T)$$

An example for a type is $\mathbb{N}$ as a name for the set of all natural numbers. □

Messages are data. By concatenation of messages we get streams of messages. We represent streams as functions that map an index to the message at this index in the stream. The set of messages that occur in a stream determines the type of the stream.

DEFINITION 2 (STREAM):
Let M be a set of messages. By M^* we denote the set of finite streams of messages over M; M^∞ is the set of infinite streams over M and $M^\omega = M^* \cup M^\infty$.
We represent streams as functions:

$$\begin{aligned} s : [1 \ldots n] \to M \quad &\Leftrightarrow s \in M^* \\ s : \mathbb{N}_+ \to M \quad &\Leftrightarrow s \in M^\infty \end{aligned}$$

Occasionally, we represent a stream s of n messages as a construct $\langle m_1, m_2, \ldots, m_n \rangle$ such that $\forall\, i \leq n : s(i) = m_i$. By $\langle\rangle$ we denote the empty stream. □

A stream carries no timing information. Therefore, we introduce timed streams that consist of consecutive time intervals; each interval represents the finite streams of messages that appear during this interval.

DEFINITION 3 (TIMED STREAM):
We use $M^{\underline{*}}$, $M^{\underline{\infty}}$ and $M^{\underline{\omega}}$ respectively to denote finite, infinite, and both types of timed streams.
We represent timed streams as functions:

$$\begin{aligned} s : [1 \ldots n] \to M^* \quad &\Leftrightarrow s \in M^{\underline{*}} \\ s : \mathbb{N}_+ \to M^* \quad &\Leftrightarrow s \in M^{\underline{\infty}} \end{aligned}$$

Timed streams are arrays of untimed streams. We use double angle brackets $\langle\!\langle\rangle\!\rangle$ to delimit timed streams. □

Channels are connections between communicating entities and carry information in terms of streams. A system's input- and output channels characterize the system's interface and determine its communication facilities. Channels are typed and fix the kind of messages that may appear in the communicated stream.

Usually a system has more than one channel to communicate with the environment. Therefore, we need to talk about sets of channels and the streams of messages that appear on them over time.

DEFINITION 4 (CHANNEL HISTORY):
A channel has a name. Let C be a set of channel names. A channel history x describes the communication of messages on a set of channels over time and is a function

$$x : C \to (\mathbb{N}_+ \to \mathbb{D}^*)$$
$$x : C \to ([1 \ldots n] \to \mathbb{D}^*)$$

for infinite and finite channel histories respectively. By $\vec{C}$ as well as by $\mathbb{H}(C)$ we denote the set of all possible type correct channel histories over a set of channels C. By $\mathbb{H}_n(C)$ we denote channel histories that have length n over the set of channels C. If we explicitly require infinite channel histories we refer to them as $\mathbb{H}_\infty(C)$. □

Note: In FOCUS a timed stream always is infinite because time ever evolves even if no more messages appear. A timed stream has an infinite number of empty time intervals after the last communicated message. In this thesis we will deal with partial specifications. This includes specifications addressing only a limited time scope. Therefore, we explicitly allow finite timed streams as well. Similar, we allow finite channel histories.

In the context of channel histories the idea of timed streams becomes important. Time intervals allow relating the occurrence and order of messages on different channels.

EXAMPLE 2.1 (TIMING CORRELATION OF MESSAGES)
Regard the two (untimed) streams of natural numbers:

$$\langle 1, 4, 7, 8, 3, \ldots \rangle$$
$$\langle 2, 8, 14, 16, 6, \ldots \rangle$$

In these streams it is impossible to conclude if message 4 in the upper stream appears before, simultaneously or after message 8 in the lower stream. The same messages appearing in a timed stream allow this conclusion:

$$\langle\!\langle \langle\rangle \quad \langle 1 \rangle \qquad\quad \langle 4 \rangle \ \langle\rangle \ \langle\rangle \ \langle 7, 8, 3 \rangle \ \ldots \rangle\!\rangle$$
$$\langle\!\langle \langle 2 \rangle \ \langle 8, 14, 16, 6 \rangle \ \langle\rangle \quad \langle\rangle \ \langle\rangle \ \langle\rangle \qquad \ldots \rangle\!\rangle$$

In this example message 4 in the upper stream is communicated after message 8 in the lower stream. ♣

2.2. Auxiliary functions on streams and channel histories

Subsequently we introduce some useful functions on streams and channel histories that help defining operations on the semantic domain. Note that we overload some of the function symbols. By looking at the signature one can figure out the used version of the function.

First we introduce a function that returns the length of a stream, both, for untimed and for timed streams.

$$
\begin{aligned}
&\#: M^{\omega} \to \mathbb{N}, \\
&\quad s \mapsto k \quad \Leftrightarrow \quad s \in [1 \ldots k] \to M \\
&\quad s \mapsto \infty \quad \Leftrightarrow \quad s \in \mathbb{N}_{+} \to M \\
&\#: M^{\underline{\omega}} \to \mathbb{N}, \\
&\quad s \mapsto k \quad \Leftrightarrow \quad s \in [1 \ldots k] \to M^{*} \\
&\quad s \mapsto \infty \quad \Leftrightarrow \quad s \in \mathbb{N}_{+} \to M^{*}
\end{aligned}
$$

The concatenation assembles two streams. Note that the concatenation of some stream to an infinite stream has no effect.

$$
\circ: M^{\omega} \times M^{\omega} \to M^{\omega}, \quad r \circ s \mapsto t \quad \Leftrightarrow \quad t(k) = \begin{cases} r(k), & \text{if } 1 \leq k \leq \#r \\ s(k - \#r), & \text{if } \#r < k \leq (\#r + \#s) \end{cases}
$$

In a similar way, we define the concatenation of timed streams. Usually, we index the timed variant with a "t" but occasionally drop the index if the variant is clear from the context.

$$
\circ_t: M^{\underline{\omega}} \times M^{\underline{\omega}} \to M^{\underline{\omega}}, \quad r \circ_t s \mapsto t \quad \Leftrightarrow \quad t(k) = \begin{cases} r(k), & \text{if } 1 \leq k \leq \#r \\ s(k - \#r), & \text{if } \#r < k \leq (\#r + \#s) \end{cases}
$$

We generalize the concatenation of streams to the concatenation of channel histories. However, this needs some additional considerations: if streams in the channel history have different lengths we fill them with empty time intervals such that all streams become of equal length. We append the second channel history point-wise at the end of those streams.

$$
\circ_t: \mathbb{H}(C) \times \mathbb{H}(C) \to \mathbb{H}(C),
$$

$$
\begin{aligned}
&\text{Let } l = max(\#(r_1), \#(r_2), \ldots, \#(r_n)) \text{ in}\\
&(c_1 \mapsto r_1, c_2 \mapsto r_2, \ldots, c_n \mapsto r_n) \circ_t (c_1 \mapsto s_1, c_2 \mapsto s_2, \ldots, c_n \mapsto s_n) \mapsto\\
&\qquad (c_1 \mapsto (r_1 \circ_t (\langle\rangle)^{(l-\#(r_1))} \circ_t s_1),\\
&\qquad\ c_2 \mapsto (r_2 \circ_t (\langle\rangle)^{(l-\#(r_2))} \circ_t s_2), \ldots,\\
&\qquad\ c_n \mapsto (r_n \circ_t (\langle\rangle)^{(l-\#(r_n))} \circ_t s_n))
\end{aligned}
$$

As two special cases, we introduce the concatenation of channel histories of length one with channel histories of arbitrary length. For a better distinction, we introduce new function symbols. This allows us to use pattern matching to split up a stream into a head and a tail or into a leading channel history and a last single time interval. Let C be a set of channels and $\mathbb{H}_1(C)$ the set of channel histories over the channels C with a length of exactly one time interval ($\forall\, c \in C : \#(c) = 1$).

$$
\begin{aligned}
&\gg : \mathbb{H}_1(C) \times \mathbb{H}(C) \to \mathbb{H}(C),\\
&\quad c_1 \gg c = c_1 \circ_t c\\
&\ll : \mathbb{H}(C) \times \mathbb{H}_1(C) \to \mathbb{H}(C),\\
&\quad c \ll c_1 = c \circ_t c_1
\end{aligned}
$$

With the concatenation of (un)timed streams and channel histories at hand we are able to define the prefix relation for (un)timed streams as well as for channel histories:

$$
\begin{aligned}
&\sqsubseteq\colon M^{\omega} \times M^{\omega} \to \mathbb{B},\\
&\quad s \sqsubseteq r \quad \Leftrightarrow \quad \exists\, t \in M^{\omega} : s \circ t = r\\
&\sqsubseteq\colon M^{\underline{\omega}} \times M^{\underline{\omega}} \to \mathbb{B},\\
&\quad s \sqsubseteq r \quad \Leftrightarrow \quad \exists\, t \in M^{\underline{\omega}} : s \circ_t t = r\\
&\sqsubseteq\colon \vec{C} \times \vec{C} \to \mathbb{B},\\
&\quad s \sqsubseteq r \quad \Leftrightarrow \quad \exists\, t \in \vec{C} : s \circ_t t = r
\end{aligned}
$$

Now we are able to define the truncation of timed streams at a certain time. Let s be a timed stream and $t \in \mathbb{N}$.

$$
\downarrow\colon M^{\underline{\omega}} \times \mathbb{N} \to M^{\underline{\omega}}, \quad (s)_{\downarrow n} \mapsto s' \quad \Leftrightarrow
$$

$$
s' = \begin{cases} s & \text{if } n \geq \#s \\ \langle\!\langle\rangle\!\rangle & \text{if } n < 1 \\ r \mid r \sqsubseteq s \wedge \#r = n & \text{otherwise} \end{cases}
$$

The time truncation until time t returns a timed stream that contains all time intervals until and including time interval t. It easily generalizes to channel histories. Let c be a channel history over channels C.

$$\downarrow\colon \vec{C} \times \mathbb{N} \to \vec{C}, \quad (c)_{\downarrow t} \mapsto c' \quad \Leftrightarrow$$
$$\forall x \in C : c'.x = (c.x)_{\downarrow t}$$

The time truncation starting at time t returns a timed stream that contains all the time intervals starting after time t until the end of the given stream s. The time interval t itself is excluded.

$$\uparrow\colon M^{\underline{\omega}} \times \mathbb{N} \to M^{\underline{\omega}}, \quad (s)^{\uparrow t} \mapsto s' \quad \Leftrightarrow$$
$$s' = \begin{cases} s & \text{if } t \leq 1 \\ \langle\!\langle\rangle\!\rangle & \text{if } t > \#s \\ r \mid \exists n : \#(n) = t \wedge n \circ_t r = s & \text{otherwise} \end{cases}$$

Again the time truncation at time t easily generalizes to channel histories. Let c be a channel history over channels C.

$$\uparrow\colon \vec{C} \times \mathbb{N} \to \vec{C}, \quad (c)^{\uparrow t} \mapsto c' \quad \Leftrightarrow$$
$$\forall x \in C : c'.x = (c.x)^{\uparrow t}$$

Let s be a timed stream and c be a channel history. We define an arbitrary clipping of a timed stream and a channel history respectively as:

$$s[m \ldots n] \stackrel{def}{=} ((s)_{\downarrow n})^{\uparrow m-1}$$
$$c[m \ldots n] \stackrel{def}{=} ((c)_{\downarrow n})^{\uparrow m-1}$$

The interval function returns a stream that contains exactly the time intervals between and including time m and n.

In addition to the concatenation of streams we define a function that pastes streams. The function is partial.

$$\frown\colon \vec{C} \times \vec{C} \to \vec{C}, \quad c_1 \frown c_2 \mapsto c_3 \quad \Leftrightarrow$$
$$(c_3)_{\downarrow \#(c_1)} = c_1 \wedge (c_3)^{\uparrow \#(c_1)-1} = c_2$$

The function takes two channel histories and combines them such that the last time interval of the first one and the first time interval of the second one *overlap*.

A channel restriction selects a subset of channels of a channel history. Let C_1 and C_2 be two sets of channels and $C_3 = C_1 \cap C_2$

$$\lfloor: \mathbb{H}(C_1) \times C_2 \to \mathbb{H}(C_3), \quad x_{\lfloor C_2} \mapsto x' \quad \Leftrightarrow$$
$$\forall c \in C_3 : x(c) = x'(c)$$

As an inverse operation, the combination of channel histories extends the set of channels of a channel history. The combination of two channel histories creates a new channel history comprising the two former ones. Let $C_3 = C_1 \cup C_2$.

$$\bowtie: \mathbb{H}(C_1) \times \mathbb{H}(C_2) \to \mathbb{H}(C_3), \quad c_1 \bowtie c_2 \mapsto c \quad \Leftrightarrow$$
$$c_{\lfloor C_1} = c_1 \wedge$$
$$c_{\lfloor C_2} = c_2$$

Note that this is a partial function. If $C_1 \cap C_2 \neq \varnothing$ and $\exists x \in C_1 \cap C_2$ such that $c_1.x \neq c_2.x$ no c exists that satisfies the function specification.

2.3. Characterizing channel histories

Throughout the thesis we will characterize *sets* of channel histories or denote particular channel histories. Therefore, we present means of characterizing channel histories based on regular expressions and finite automata. Both are equi-expressive and can be transformed to each other (see, e.g., [Berry and Sethi, 1986]).

For our purposes, the expressiveness of regular languages is sufficient. Both parse a channel history and accept it, if it fits to the characterization. Readers who are familiar with regular expressions and finite automata safely can skip this sub-section and proceed with Section 2.4. However, we suggest having a look at Example 2.2 to get familiar with the particular syntax used in this thesis.

Regular expressions allow writing concise characterizations but become unreadable if the characterization is complex. Finite automata allow writing complex characterizations in a clear manner but are more expansive.

Both, regular languages and finite automata use an alphabet. Let $C = \{c_1, \ldots c_n\}$ be a set of channels with types $TYPE(c_1) = T_1, \ldots, TYPE(c_n) = T_n$. The alphabet of the regular language is the set of tuples

$$\Sigma = (c_1 \to T_1^*, \ldots, c_n \to T_n^*)$$

We write the empty untimed stream as "$\langle\rangle$" and an arbitrary, type correct element as "$\langle\bullet\rangle$".

A tuple $(\mathbb{Q}, \Sigma, \delta, q_0, \mathbb{F})$ defines a finite automaton with the usual meanings

- $\mathbb{Q}$ is a set of states
- Σ is the input alphabet
- q_0 is the initial state

- $\mathbb{F}$ is the set of final states
- δ is the transition relation $\mathbb{Q} \times \Sigma \to \mathbb{Q}$

The regular expressions are likewise straight forward. However, in conventional regular expressions, the operators are only applied to elements (or sequences of elements) of the alphabet. In the context of channel histories we allow applying the operators to a subset of the channels as well including a nesting of expressions. For an example of this nesting see Example 2.2.

We use the following operators to describe the language. Let α and β be two expressions describing sets of channel histories each:

Expression	Language	
ϵ:	$\{\{\!\langle\rangle\!\}\}$	
$\langle a \rangle$:	$\{\{\!\langle a \rangle\!\}\}$ for $a \in \Sigma$	//Single message defining a stream
$\{\!\langle\langle a \rangle\rangle\!\}$:	$\{\{\!\langle a \rangle\!\}\}$ for $a \in \Sigma$	//Stream containing a single message
$\alpha\beta$:	$\{a \circ_t b \mid a \in L(\alpha) \wedge b \in L(\beta)\}$	// Concatenation of α and β
$\alpha \vert \beta$:	$L(\alpha) \cup L(\beta)$	//Alternatives
$(\alpha)^*$:	$\{\} \cup \{a \circ_t b \mid a \in L(\alpha) \wedge b \in L((\alpha)^*)\}$	//repetition $[0..\infty]$
$(\alpha)^+$:	$L(\alpha) \cup \{a \circ_t b \mid a \in L(\alpha) \wedge b \in L((\alpha)^*)\}$	//repetition $[1..\infty]$
$(\alpha)^n$:	Inductively: $(\alpha)^0 = \{\{\!\langle\rangle\!\}\}$, $(\alpha)^n = \{a \circ_t b \mid a \in L(\alpha) \wedge b \in L((\alpha)^{n-1})\}$	//exactly n times
$(\alpha)^{[n,m]}$:	$\bigcup_{n \le i \le m} L((\alpha)^i)$	//at least n, at most m times

To allow for concise and intuitive channel history characterizations, we introduce a complement operator: let T be a type. For some $A \subseteq T$ we denote the complement as

$$\overline{A} \stackrel{def}{=} T \backslash A$$

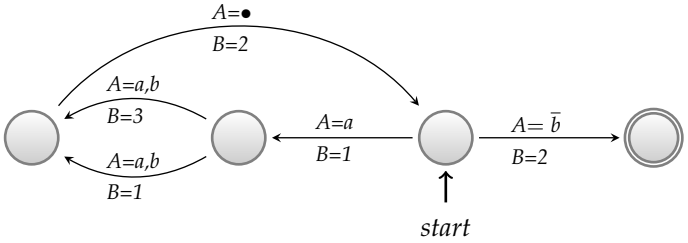
The complement is easily applied to channel histories with multiple channels by pointwise application.

If we refer to *instances* of channel histories in examples we use simple tables – especially if we point at temporal aspects. Each column corresponds to a distinct time interval. Each row contains the information sent on exactly one channel. We use versions that list the time intervals in the first row if we refer to certain times and versions without this information if only the relative timing of messages matters.

EXAMPLE 2.2 (REGULAR EXPRESSIONS FOR CHARACTERIZING CHANNEL HISTORIES) *A and B are channels with* $TYPE(A) = \{a, b, c\}$ *and* $TYPE(B) = \{1, 2, 3\}$. *A characterization of a channel history in regex-notation:*

$$A \quad \langle\!| \left(\langle a \rangle \quad \langle a, b \rangle \quad \langle \bullet \rangle \right)^* \quad \overline{b} |\!\rangle$$
$$B \quad \langle\!| \left(\langle 1 \rangle \quad \langle (1|3) \rangle \quad \langle 2 \rangle \right) \quad 2 |\!\rangle$$

The same set of channel histories in the finite automata notation:



Both define a set of I/O-histories containing, e.g., the history

t	1	2	3	4	5	6	7
A	a	a,b	a	a	a,b	b	c
B	1	3	2	1	1	2	2

♣

2.4. Stream processing functions

A discrete system is a delimited entity that has a defined interface. It communicates with its environment only via that interface. We characterize systems by the observations that we make at the interface. This is a strong abstraction because we make no assumptions about the internal structure or other implementation details. We call this perspective a black-box view.

The black-box view allows focusing on the functional issues of a system before dealing with architecture or implementation details. This is appropriate for a system specification that characterizes all acceptable implementations [Heitmeyer et al., 1996].

We decompose the overall behavior of a system into smaller functional parts called service. Similar to a system we characterize a service by an interface and the observations at that interface.

The observations at the interface of a system or a service are characterized by a relation from typed messages that the system/service receives to typed messages it generates as a response. We call the relation an I/O-behavior [Broy and Stølen, 2001].

DEFINITION 5 (SYNTACTIC INTERFACE):
Let I be a set of input channels and O be a set of output channels of the system. The tuple (I, O) represents the system's syntactic interface. We denote the syntactic interface as $(I \triangleright O)$. For all $c \in (I \cup O)$ the communicated messages m need to be of the corresponding type. By $\mathbb{I}$ we denote the set of all interfaces. □

A service is a projection of a system or another service on a sub-interface. Therefore, we syntactically relate interfaces.

DEFINITION 6 (SUB-INTERFACE):
For a service interface $(I \triangleright O)$ we say that $(I' \triangleright O')$ is a sub-interface written as $(I' \triangleright O') \subseteq (I \triangleright O)$ iff

$$I' \subseteq I \wedge O' \subseteq O \wedge TYPE(I') \subseteq TYPE(I) \wedge TYPE(O') \subseteq TYPE(O)$$ □

The definition of a syntactic interface determines the messages that a system/service communicates at its interface. The actual relation of this messages is the semantic interface.

DEFINITION 7 (SEMANTIC INTERFACE):
We call the behavior of a system/service its semantic interface:

$$F : \vec{I} \rightarrow \mathcal{P}(\vec{O})$$

We talk about an I/O-behavior and use $\mathbb{F}(I \triangleright O)$ to denote the set of all I/O-behaviors over the interface $(I \triangleright O)$. □

In general, a single input history maps to a set of output histories to represent non-determinism. We allow for partial functions and for total functions. Therefore, we define domain and range for functions in accordance to [Broy, 2005].

DEFINITION 8 (DOMAIN AND RANGE OF STREAM PROCESSING FUNCTIONS):
We define the domain of a function F – denoted as $dom(F)$ – and the range of a function – denoted as $ran(F)$ – as:

$$dom(F) = \{s \in \vec{I} \mid F.s \neq \varnothing\}$$
$$ran(F) = \{r \in \vec{O} \mid s \in dom(F) \wedge r \in F.s\}$$ □

Similar to the sub-interface relation that syntactically relates two interfaces, we introduce slicing to relate behaviors.

DEFINITION 9 (SLICE):
Let $S \in \mathbb{F}(I \rhd O)$ be a service and $(I' \rhd O')$ be a sub-interface of $(I \rhd O)$. We define a slice of S with respect to $(I' \rhd O')$ – denoted as $S\dagger(I' \rhd O')$ – as:

$$S\dagger(I' \rhd O').(x_{\lfloor I'}) = \{y_{\lfloor O'} \mid y = (S.x)\}$$ □

All realizable systems need to satisfy an important property: *causality* [Broy and Stølen, 2001]. Causality requires any output only to depend on earlier inputs. In the context of I/O-behaviors, nobody prevents a system designer from specifying behaviors that depend on future inputs. Of course such a system is unrealizable.

DEFINITION 10 (CAUSAL I/O-BEHAVIOR):
We call an I/O-behavior F weakly causal, iff

$$\forall x, z \in \vec{I}, t \in \mathbb{N}:$$
$$(x)_{\downarrow t} = (z)_{\downarrow t} \Rightarrow \{(y)_{\downarrow t} : y \in F(x)\} = \{(y)_{\downarrow t} : y \in F(z)\}$$

i.e., outputs at time t depend only on inputs until time t. We call the I/O-behavior strictly causal, iff

$$\forall x, z \in \vec{I}, t \in \mathbb{N}:$$
$$(x)_{\downarrow t} = (z)_{\downarrow t} \Rightarrow \{(y)_{\downarrow t+1} : y \in F(x)\} = \{(y)_{\downarrow t+1} : y \in F(z)\}$$

i.e., outputs at time $t + 1$ depend only on inputs until time t. This considers that calculations require some time to happen. □

Subsequently, we mainly use strict causality and occasionally drop the "strictly". For a discussion about differences of weak and strict causality, especially in the context of compositions, see [Broy and Stølen, 2001].

Note that stream processing functions work without states. States are an encoding for an equivalence class of channel histories. To determine the outputs of a system at some time t it is equivalent a) to regard the complete input history and the complete output history (for non-deterministic systems) until time $t - 1$ or b) to use a state and just the current input to calculate the next output. [Broy, 2007] explains the relation of channel histories and abstract state machines.

Specifying directly in the semantic domain of I/O-histories is cumbersome. Therefore, [Broy and Stølen, 2001] presents a number of specification techniques. These specification techniques include inter alia state machines, functional notations, and assumption guaranty specifications.

Chapter 3

Context adaptation

Context-adaptive systems liberate users from some interactions with the system by considering the proximate environment for providing appropriate functionality. During the past years a number of definitions emerged concerning the additional use of information about the proximate environment – most of them in the context of proposed architectures and algorithms. Many definitions use "context information" to distinguish between conventional systems and context-adaptive systems but miss to discuss the determination of the context information.

We present a formal definition of the terms "context-aware system" and "context-adaptive system" together with related terms in Section 3.1. This definition is the basis of any further considerations about formal specifications. Subsequently, we discuss variants of context-adaptive systems and differences in the goals they serve. We give a taxonomy of context-adaptive systems with respect to three dimensions (user dimension, system dimension, and flexibility dimension) in Section 3.2. This taxonomy allows us to classify the MARLIN approach and to demarcate it from other approaches that serve different goals. As a conclusion of this chapter we discuss prior definitions and related terms and concepts in Section 3.4.

Contents

3.1. Context adaptation and I/O-relations

The idea of context-adaptive systems is simple: systems infer about selected inputs and choose different observable behaviors accordingly. The spectrum of descriptions of such systems in literature is broad. At the one end of the spectrum there are simple systems like an automatic light that is sensitive to movements and has a simple algorithm. At the other end of the spectrum there are systems that use artificial intelligence and machine learning for the implementation of complex inference mechanisms.

All of these systems adapt their behavior in accordance with selected (possibly interpreted) information, aggregated knowledge, and possibly a cost function. We find systems with these properties in history even at times when the terms "context-aware-" and "context-adaptive systems" were not yet coined.

Although the intuition behind context-adaptive systems seems clear, defining context-adaptive systems is difficult. Liberating users from interactions requires the automation of reactions to changes in the environment on behalf of the user [Dey et al., 2001; Chen and Kotz, 2000; Schilit et al., 1994]. However, all computer systems show some automatisms. Without them they hardly are of any use.

We base our definitions of context-aware and context-adaptive systems on earlier work that we presented in [Broy et al., 2009] and apply some extensions and changes. The original idea is to utilize a reference system that communicates with the system under construction as an instance that assists the classification of behaviors as adaptive.

3.1.1. Context-adaptive systems

The term "context" describes the central concept of context-adaptive systems. Speaking of context in general is unspecific without relating it. Without reference, everything outside the system under construction is context which is of no use for the definition or during development.

We relate the term "context" to the interaction between a user and the system under construction i.e. the conditions of use. In this thesis the term "user" may refer to a human user as well as to some other technical system.

Note that this choice allows selecting different aspects as context for specific applications:

The physical environment of the system may become a part of the considered context if the system's physical environment is part of the usage context – even if the user is distant to the system (e.g., connected remotely). An example is a mars lander. The physical environment of the user (here the operator) is unimportant. In this application the physical environment of the mars lander is important.

A user as the reference allows applying the definition to systems whose physical environment is insignificant and rather information about the users' physical environment is provided remotely (e.g., in cloud computing). As an example regard a system that receives data about users remotely to provide location based services via a web service. The physical environment (e.g., location) of the system is unimportant but the physical environment of the user (here human user) is important.

We model a user as another system U that communicates with the system under construction S. This communication is a direct and intentional communication of the user. We regard any other interactions with the system as context to the interaction between U and S. These interactions are indirect and unintentional from the user's point of view. We use C to represent this context.

For the development of systems that liberate users from interactions we regard it as a natural choice to use a dedicated user model to acquire the user needs and requirements and to design the communication between system and user. In this thesis we require the existence of a user model (or at least its syntactic interface) but exclude the discussion about its generation. The actual behavior of the user is unimportant for the definition of the term "context-adaptive systems" but it is useful and necessary for some kinds of analysis (see, e.g., [Winter, 2009]). In this thesis we only use the syntactic interface between both.

Figure 3.1 shows a system S that interacts with a user U and a context C. Subsequently we define the channels appearing in the figure.

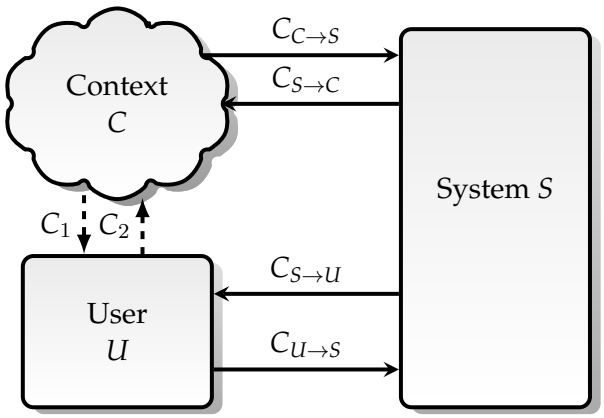


Figure 3.1.: User, system, and their context

DEFINITION 11 (USER INTERFACE):
Let $(I \triangleright O)$ be the system S under development. We anticipate a user U and define the channels for the communication with the system under development. We call these channels the user interface $U = (I_U \triangleright O_U)$.

We denote channels from the user to the system by $C_{U\to S} = I \cap O_U$ and call them the user input interface (user inputs for short). We also refer to inputs from the user as direct inputs. Channels from the system to the user $C_{S\to U} = O \cap I_U$ are the user output interface or user outputs for short.

The user interface defines the possible interactions between system and user. □

A system usually communicates with entities different from the user as well. From the user's perspective these other entities are in the context.

DEFINITION 12 (CONTEXT INTERFACE):
We call the channels $C_{C\to S} = I \backslash C_{U\to S}$ the context input interface. These channels carry the information that is the context to the interactions between the user and the system.

Formally there are channels $C_{S\to C} = O \backslash C_{S\to U}$ for the communication with the environment other than the user. This is the context output interface. However, it is irrelevant for the definitions. □

The user might interact with the elements in the context as well (intentionally and/or unintentionally). This interaction can be a context to the interaction between user and system as well.

A system is context adaptive if it uses additional information (context information) to change the behavior that is observable at the user interface. Therefore, it acts and reacts different even if the user inputs are equal.

DEFINITION 13 (CONTEXT-ADAPTIVE SYSTEM):
We consider the slice $S^U = S\dagger((C_{U\to S}) \rhd (C_{S\to U}))$. A system S is context-adaptive iff

$$\exists\, i_1, i_2 \in dom(S) :$$
$$i_1{\lfloor_{C_{U\to S}}} = i_2{\lfloor_{C_{U\to S}}} \wedge (S.i_1){\lfloor_{C_{S\to U}}} \neq (S.i_2){\lfloor_{C_{S\to U}}}$$ □

The set of possible reactions with respect to the user output interface differ ($S.i_1{\lfloor_{C_{S\to U}}} \neq S.i_2{\lfloor_{C_{S\to U}}}$) even for the same user inputs ($i_1{\lfloor_{C_{U\to S}}} = i_2{\lfloor_{C_{U\to S}}}$). The contextual inputs determine the actual outputs. Indeed, the inputs i_1 and i_2 are different but their difference is only observed at channels $C_{C\to S}$.

Speaking of system behaviors, we talk about (complex) functions that map inputs to outputs. Context is just another input to such functions. It influences the reactions of a system just like any other input does. Nevertheless, for creating specifications it is useful to take different viewpoints.

From the viewpoint of a user (respectively a user model) a system behavior may change with respect to inputs that are not made directly by the user. In contrast, a comprehensive observer is able to relate all observations at the system's output interface to the respective inputs including those that have a different source than the user. Therefore, we distinguish two different ways to describe a system:

Adaptive viewpoint The system specification composes different behaviors that users can observe in different contexts. The system selects the behaviors using (additional) contextual information. This view leads to specifications of systems that take contextual inputs as an argument and return usage functions that interact with the user.

Non-adaptive viewpoint The system's specification bases on a set of (complex) functions that do not distinguish between user inputs and contextual inputs. A possible structuring principle in these specifications is a hierarchical decomposition of the interface (see, e.g., slicing in Section 5.2.2) without special considerations about the way a user interacts with the system.

Context-adaptive systems are a matter of the viewpoint and just lead to a certain structure of the specification. We can say that context adaptation is a design pattern. The structure encourages the designers to think about the usage of the system explicitly rather than focusing only on inputs and outputs. They can consider the usage of the system in different situations in isolation and structure the system specification accordingly.

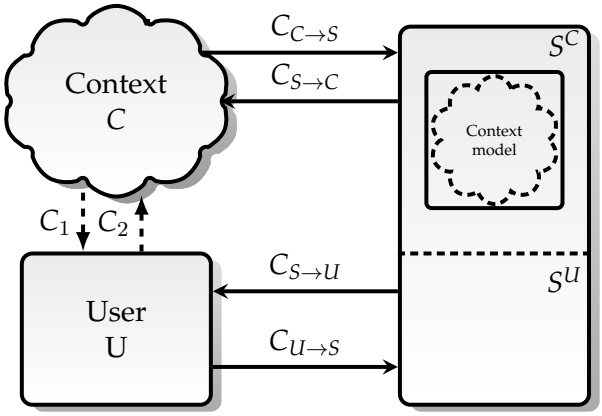


Figure 3.2.: Context-adaptive system with two sub-systems. One offers functions to the user, the other controls the selection of the offered function

Figure 3.2 illustrates the relation between user, system, and context. Defining the user interface virtually divides the system into two sub-systems: the system core (denoted as S^U) represents the usage functions of the system the user interacts with. The adaptation sub-system (denoted as S^C) capsules the logic that controls the core system. It is responsible for choosing proper usage functions. Obviously, S^U and S^C need to interact to offer the intended overall system behavior.

Some systems fit to the definition of context-adaptive systems but may not be regarded as such. A simple example is a conventional browser: Depending on an available network connection the browser either displays the requested web site or an error message: for equal inputs (the same URL) the user observes two different reactions. Most people refuse calling such a conventional browser context-adaptive.

More people are likely to call the browser context-adaptive if we consider a browser that adapts video stream encodings according to the available bandwidth. Both browsers fit to the definition and their actual consideration as context-adaptive systems depends on the attitude of the observer. From a formal point of view, both, the availability of a network connection and the actual bandwidth have sources different from the user and determine the actual interaction with the user.

The actual use of the definition is a methodological one for structuring the specification, e.g., into

1. entities that represent the behavior during phases of execution in the core system
2. a logic that selects the behaviors in dependence of contextual inputs

It is possible to split the specification of the bandwidth-aware browser, e.g., into a core system with two behaviors: one for low and another for high bandwidth and an adaptation sub-system that selects a behavior depending on the bandwidth. Later, e.g. during system design, the notion of context-adaptive systems allows the elaboration of an architecture with a flexible linking of components (like, e.g., presented in [Fahrmair et al., 2006a]) to overcome bandwidth shortcomings.

3.1.2. Context model, context, and situation

In literature the terms context and context model are used in many different ways. We regard a context model as an abstraction of the environment to those aspects that are relevant for the interaction between user(s) and system. That is why we call it a "model". A context model defines the state space that allows characterizing situations.

DEFINITION 14 (CONTEXT MODEL):
Formally, we define a context model as a special purpose data model. The data model describes all possible combinations of information that the system observes in addition to interactions with the user.

In addition we relate the data model already to a set of channels since the system will observe instances of the data model in terms of combinations of events at these channels.

Let WORLD be the universe of all aspects of the observable reality, CANNELS be the set of all channels, and S the system under development. To emphasis the special purpose of the context model we represent it by a function [].

Formally the relation has the signature

$$[\,] : (\{S\} \to \mathcal{P}(WORLD)) \to \mathcal{P}(CHANNELS),$$
$$S \mapsto C \Rightarrow C \subset I_S \cup O_S \qquad \square$$

where I_S and O_S are the input- and output channels of the system S that is under construction.

Note that the argument to the function [] *is the system under construction. The context model is specific for the system i.e. different systems will have different context models.*

Figure 3.2 shows the context model. It represents relevant aspects of the reality within the system.

EXAMPLE 3.1 (CONTEXT MODEL)
A system whose interaction with the user is influenced by the weather could use a context model that maps the aspect weather to a representation that considers two inputs i_1 and i_2 with

$$type(i_1) \stackrel{def}{=} \{sunny, cloudy, raining\} \qquad //weathercondition$$
$$type(i_2) \stackrel{def}{=} [-40;\ 50] \qquad //temperature \quad \clubsuit$$

Note, the term context model only defines the possible observations that are relevant for the interaction. The term context (cf. Definition 15) captures the actual dependencies between events.

The context model abstracts from any specific sensor technology. The abstraction even may result in information that no sensor can measure directly. The system infers the information from multiple sensor data instead. For a functional specification this abstraction is appropriate because it allows focusing on the function instead of the information acquisition. A well-chosen abstraction supports the reuse of a system.

EXAMPLE 3.2 (ABSTRACTION FROM SENSOR TECHNOLOGIES)
As an example in the case study (cf. Section 1.5) we use the information about the position of the user. We abstract from the actual detection technology (e.g., by motion detectors or computer vision or simply the proximity of an RFID-tag). ♣

The simple appearance of the definition hides the importance of the context model and the efforts behind defining an appropriate one. The context model defines the state space to define contexts. The elaboration of the context model includes identifying information (including abstract types, interpretations, and procession) that is appropriate for identifying the relevant situations. Errors in the context model result in a system's inability to recognize situations properly causing unwanted behaviors (UB) [Fahrmair et al., 2006b] or automation surprises [Palmer, 1995; Sarter et al., 1997]. Sitou et al., e.g., present an approach for the methodological elaboration of context models [Sitou and Spanfelner, 2007; Sitou, 2009].

While a context model defines a state space, contexts are (sets of) instances or valuations of the state space. A context is characterized by properties over the context model. In [Sitou, 2009], Sitou presents an approach for the coupled elicitation of requirements and the contexts they are effective in. Sitou defines contexts as phases during which the contextual inputs satisfy a certain condition. We call these contexts *simple contexts*.

In the thesis at hand we give a more general definition. We consider complex and possibly longer lasting sequences of events as indications for the beginning (entry condition) of a context and again complex and possibly longer lasting sequences of events as indications for the end (exit condition) of a context. We call these contexts *complex contexts*.

Complex contexts allow for using complex interactions between system and environment to define contexts. This is, e.g., useful if certain sequences and combinations of events characterize a context. Even more, this allows defining contexts that can only be identified during runtime by analyzing the interaction between system and environment. Note that the later use of context requires an additional model: a model about the behavior of the environment that allows deriving conclusions from observations. Strictly speaking, this is some kind of context model as well but is different from our definition as a pure data model – in contrast this model needs to be a behavioral model. Since we will not explicitly refer to this kind of model in this thesis we do not give it a name.

DEFINITION 15 (CONTEXT AND SITUATION):
A context, also called a situation, is a predicate on the domain that the context model defines. The predicate characterizes (classes of) observations at the context input interface. We decompose this predicate into an entry condition and an exit condition.

Let $[S]$ be a context model for a system S and let $c \in \mathbb{H}([S])$ be a channel history. A context is a tuple $(start, stop)$ such that $start : \mathbb{H}([S]) \to \mathbb{B}$ and $stop : \mathbb{H}([S]) \to \mathbb{B}$ are predicates on streams characterizing a context's start and stop conditions. If we use the regular expression syntax for characterizing the respective sets of I/O-histories, we abbreviate $(\{\!|START|\!\}, \{\!|STOP|\!\})$ as $(c \in L_{START}, c \in L_{STOP})$ for a channel history c. □

Note that the predicates defining the contexts define equivalence classes over the reality. The predicates relate to the context model. Only situations in reality that differ at aspects of the context model can be distinguished by the system. An inappropriate choice of the context model disallows the system to distinguish situations that users may want the system to react differently in. This is very closely related to the frame problem (see, e.g., [Dennett, 1984] for a discussion of the frame problem.)

We cover simple contexts by an abbreviation: Let v be a set of values for the aspects of the context model. For a simple context we write $C = \sharp(v)$ and define:

$$\sharp(v) \stackrel{def}{=} (\{\!|\langle \overline{v} \rangle^* \langle v \rangle|\!\}, \{\!|\langle v \rangle^* \langle \overline{v} \rangle|\!\})$$

Informally spoken: as long as the system receives inputs according to v, it considers the context C as active.

Usually the context model directly relates to channels $C_{C \to S} \cup C_{S \to C}$. However, in some cases it is useful to include user inputs or outputs for defining situations. In such cases, the context model $[N]$ additionally relates to channels $C_{U \to S}$ and $C_{S \to U}$. Nevertheless, we always require that $[N] \cap C_{C \to S} \neq \varnothing$ i.e. the context model always accounts for the context input interface.

Figure 3.3 illustrates START and STOP conditions with respect to a channel history of input and output channels. The conditions' scopes are the whole history up to the current time. This allows writing conditions that capture complex correlations in the context and interactions between system and context.

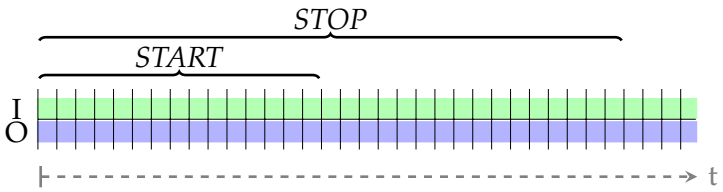


Figure 3.3.: START and STOP conditions relating to a input/output channel history

---

EXAMPLE 3.3 (CHARACTERISTICS OF CONTEXTS)

Example a) *Referring to Example 3.1 one reasonable definition for a context "bad weather" are the conditions*

$$start \stackrel{def}{=} (i_1 = cloudy \wedge i_2 < 10) \vee (i_1 = raining)$$
$$stop \stackrel{def}{=} (i_1 = sunny)$$ ♣

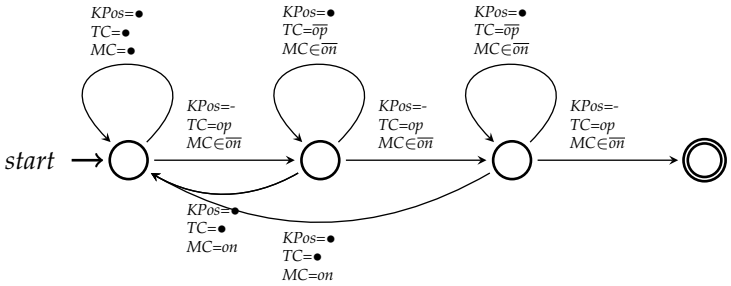
Note that the stop condition is not the inverse of the start condition. Depending on the intention it is reasonable to define such "asymmetric" conditions.

Example b) *In the case study the automatic trunk door only works as long as the battery is high. This is an example for a simple context. Based on the context model from Section 1.5 we define this context as* $\sharp(UBat = hi)$.

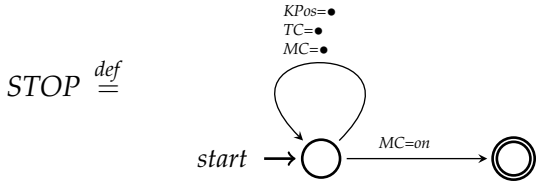
Example c) *We use the misuse protection as an example for both, a complex situation and a situation that bases partly on user inputs* [1].

Opening the trunk by gesture detection shall only work while the system detects no misuse. We define a misuse as detecting a gesture three times in a row without receiving a proper key authentication at the same time. The start of the engine resets the counter. The corresponding definition is $(\langle\!\langle\ START \rangle\!\rangle, \langle\!\langle\ STOP \rangle\!\rangle)$ *with*

$START \stackrel{def}{=}$



[1] The FOCUS approach does not allow that two different components write to the same channel. Therefore, we may regard detected gestures of some other user without a key formally as information on a different channel or use some model of an arbitration. The gesture channel then can be context for the user under consideration as well. However, for the purpose of this thesis it is sufficient to consider only one input channel.



We will use contexts to guard functionality, i.e., describe the necessary conditions for a functionality to be observable. In colloquial language the term "context" often describes both, a general condition and a current setting. The term "current context" avoids this ambiguity.

DEFINITION 16 (CURRENT CONTEXT, CURRENT SITUATION):
Starting at the time a contexts entry condition holds for the first time we say that the context is current, effective, or active until its exit condition holds.

Let $[N]$ be a context model and c be a finite channel history over $[N]$ ($c \in [\vec{N}]$) that captures the messages until the considered time. Formally, a context C is current/effective/active with respect to c (written as $c \Vdash C$) iff

$$
\begin{aligned}
&C = (start, stop) \wedge \\
&\quad \exists p, q \in [\vec{N}],\ \forall r \in [\vec{N}] : \\
&\qquad c = p \circ q && \wedge \quad \text{// c can be divided into some p and q} \\
&\qquad start(p) && \wedge \quad \text{// p satisfies the entry condition} \\
&\qquad [(r \sqsubseteq q) \Rightarrow \neg stop(p \circ r)] && \quad \text{// the exit condition is not yet satisfied after p}
\end{aligned}
$$

We furthermore say that a certain context S is effective/current/active in a phase $[t_i, t_j]$ of a channel history c if:

$$
\begin{aligned}
&\forall t' \in \mathbb{N} : \\
&\quad t_i \leq t' \leq t_t \Rightarrow (c)_{\downarrow t'} \Vdash S
\end{aligned}
$$

Figure 3.4 illustrates a phase of an active context and shows p, q, and some r from the definition. □

In general it is impossible to figure out whether a context is active or not only by looking at the current contextual inputs. This only works for simple contexts. Usually an observer needs to investigate a longer history (sometimes even the whole history – depending on the exact nature of entry and exit condition) to figure out the active context.

Because the use of the terms in colloquial language is sometimes ambiguous we summarize the three important terms:

Context model is a data model that captures the important aspects of the environment that the system observes. It is an abstraction of the environment and determines the distinguishable situations.

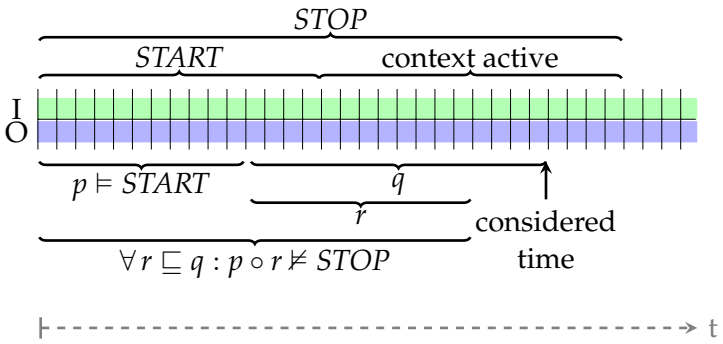


Figure 3.4.: The fragments of a input/output channel history c that definition 16 uses to define a current context

Contexts are sets of histories over the context model. Contexts are conditions that are observed as streams of valuations of the context model.

Current context (effective/active) at a time t is the context whose entry condition was satisfied before t and its exit condition not satisfied since. The term current context always refers to a certain point in time.

3.2. A taxonomy of context-adaptive systems

We kept the definitions of the preceding section as general as possible. However, context adaptation appears differently in specifications. Regarding a system as context-adaptive during a development process always *must* be an explicit decision by the system engineers. We support the understanding of consequences of specifying a system as context aware and the intuition behind the notion by classifying context-adaptive systems in three dimensions:

- the user dimension focuses on the observations made by an (artificial) user
- the system dimension classifies the systems with respect to the structuring principle that engineers apply to their specifications and
- the flexibility dimension focuses on the degree of freedom of the system

3.2.1. User dimension

We use a set of channels that the system engineers intentionally choose to be the interaction points between users and the system for the definition of context-adaptive systems. Nevertheless, a user may observe or even use additional channels. These additional channels and a user's understanding of correlations within the system individually influence the user's perception of a system as context-adaptive system.

The classification in the user dimension categorizes the way users interact with the system. The system's engineers use the subsequent classification in the user dimension in the requirements elicitation phase to identify an average user and possible consequences of divergences from the anticipated system understanding of the average user. This allows analyzing the target group with their peculiarities and respectively elaborating their requirements.

We will focus on different sets of channels subsequently and omit those channels in the figures that are of no matter (although they might exist) to ease the understanding.

3.2.1.1. Non-transparent adaptations

We call an adaptation non-transparent if a user is unable to guess the reasons for the adaptation (or to be more precise: guessing is not supported by the system)[Broy et al., 2009]. A reason can be physically unobservable contextual inputs like information coming from some infra-red sensors, reasoned information from some database, etc.

Engineers can base their work on this classification if they assume that average users ignore some contextual aspects (even if they are observable in general). Note that this assumption already may be wrong. As a consequence the mental model of users about correlations in the system possibly differs from the actual correlations which results in unwanted or unexpected behavior [Fahrmair et al., 2006b].

Non-transparent adaptation exactly matches Definition 13. The definition considers only the essential communication between user and system and ignores users observing the environment thus being able to anticipate the system's adaptations.

Without a proper manual it is likely that the user generates a wrong mental model of the behavior of the system. If a technical system has the role of a user it might seem wired to speak of a mental model of the user. However, such an artificial user finally was built by human developers and we attribute the wrong mental model to them resulting in a wrongly designed interaction between the two systems.

DEFINITION 17 (NON-TRANSPARENT ADAPTATIONS):
Let the channels be defined as above. We call an adaptation non-transparent iff:

$$\exists\, i_1, i_2 \in dom(S) :$$
$$i_1|_{\lfloor C_{U\to S}} = i_2|_{\lfloor C_{U\to S}} \wedge (S.i_1)|_{\lfloor C_{S\to U}} \neq (S.i_2)|_{\lfloor C_{S\to U}} \qquad \square$$

We regard this definition as the most general one. We always can restrict ourselves to the channels appearing in this definition because they always exist. In this base-case the system provides no arrangement that ensures the users' understanding of the adaptations. We derive the later definitions regarding more comprehensive users from this definition. Figure 3.5 illustrates the base-case.

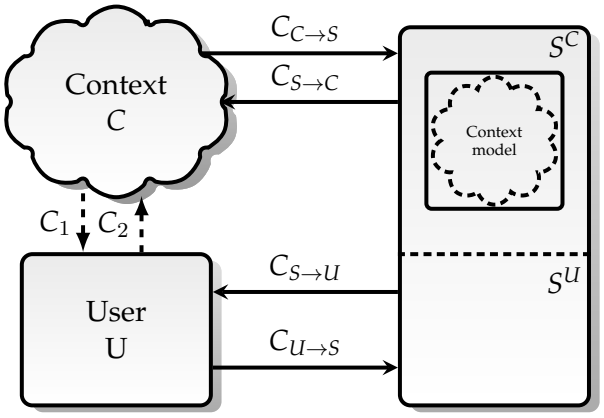


Figure 3.5.: Non-transparent adaptation: The system does not support the user to understand the adaptations, i.e. it is uncertain if the user has a correct understanding or not.

EXAMPLE 3.4 (NON-TRANSPARENT ADAPTIVE BEHAVIOR)
Consider a car navigation system that changes the calculated route to the given destination if a traffic jam is ahead. The system exhibits a non-transparent adaptive behavior if the user neither knows about the traffic jam nor does the system inform the user about the reason for a changed planning. ♣

3.2.1.2. Transparent adaptations

Non-transparent adaptation is a restricted model of adaptation. In particular, we assume the users to ignore anything relevant that happens in the environment. This is a valid assumption for technical systems in the role of an artificial user or in few domains even for human users. But human beings usually observe at least some of the relevant details of their environment and are able to build up a mental model of the system's behavior.

In addition, a context-adaptive system may tell users about the nature and reason of an adaptation to antagonize wrong expectations or surprises (i.e., the already mentioned unwanted behavior and automation surprises). Ideally, the system is context-adaptive and either is well designed thus to always perfectly meet the users' needs (usually unrealistic) or additional information allows users to understand adaptations. Either the system gives this additional information on the fly or a manual (or the like) informs the users a priori.

We call adaptations transparent, if they support users to understand adaptation decisions at least to some extent. Of course, according to the earlier discussion the level of transparency may differ. Figure 3.6 shows the general setting.

An additional channel set C_1 models the information flow from the environment to the user. The channel set C_{inf} models the information flow from the system to the user to tell the user about adaptations. Technically, the channels C_{inf} are part of the channel

set $C_{S\to U}$ but give the user additional information that make the contextual inputs on channels $C_{C\to S}$ and depending system decisions more transparent to the user.

In the formal definition we split the context interface. Channels $C_{hidden} = C_{C\to S}\backslash(C_{inf} \cup C_1)$ are still hidden to the user. Channels $C_{trans} = C_{C\to S} \cap (C_{inf} \cup C_1)$ become transparent to the user either by direct observation or additional information about adaptations that the system tells to the user.

DEFINITION 18 (TRANSPARENT ADAPTATIONS):
Let C_{inf}, C_{trans} and C_{hidden} be defined as before. The channel set $C_{Obs} = C_{U\to S} \cup C_{trans}$ describes all inputs to the system that the user either directly controls or is aware of. We call an adaptation transparent iff:

$$\{(i_1, i_2) \mid i_1, i_2 \in dom(S) \wedge i_1{\downarrow_{C_{obs}}} = i_2{\downarrow_{C_{obs}}} \wedge (S.i_1){\downarrow_{C_{obs}}} \neq (S.i_2){\downarrow_{C_{obs}}}\} \subset$$
$$\{(i_1, i_2) \mid i_1, i_2 \in dom(S) \wedge i_1{\downarrow_{C_{U\to S}}} = i_2{\downarrow_{C_{U\to S}}} \wedge (S.i_1){\downarrow_{C_{S\to U}}} \neq (S.i_2){\downarrow_{C_{S\to U}}}\}$$

Informally spoken, with the additional knowledge about contextual inputs the system is more deterministic for the user. The user uses the channels C_{trans} to distinguish some of the inputs that otherwise appear identical. □

Note that information from C_1 becomes part of the channels C_{trans}. It is dangerous to rely on these channels in general because they model only the information flow from the environment to the users without a mental model of the users. Therefore, without a decent analysis of the target users and their possible mental models, the effect of the information on C_1 remains uncertain.

We call adaptations which are deterministic with respect to channels C_{trans} fully transparent. As the definition already suggests, reality is somewhere between. A fully transparent adaptation may overload the users with information reducing the originally desired effect of removing distractions from the users. Designers may decide – after analysis of the user model and the related information on C_1 – to omit information about easy to understand adaptations or to put the information into the manual (which is not directly covered by the above definition [2]).

This does not necessarily change the specification's structure but has methodological relevance: system engineers should try to keep the channel set C_{hidden} as small as possible. Otherwise, the danger of misunderstood adaptations rises resulting in unwanted and potentially distracting behavior [Fahrmair et al., 2006b].

EXAMPLE 3.5 (TRANSPARENT ADAPTATION)
Again, consider the car navigation system. If the system tells the user about the construction, or the user hears about it on the radio, the system behavior becomes transparent adaptive. However, relying on the user hearing about the construction on the radio is uncertain and an inappropriate choice. ♣

[2] We may regard the manual as a part or the environment. Then the information in the manual becomes part of C_1

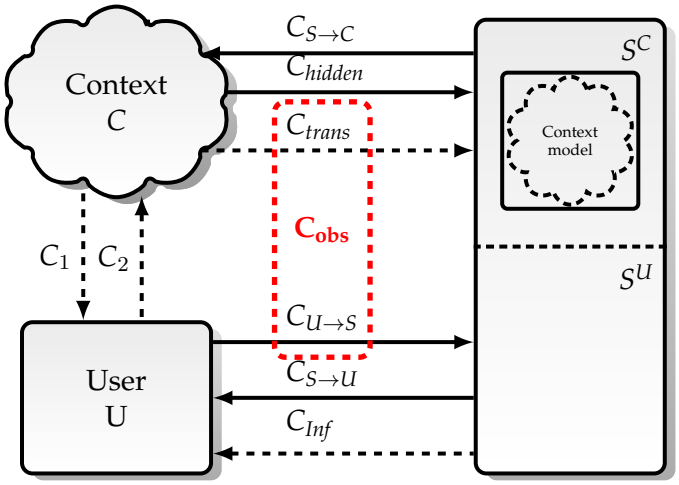


Figure 3.6.: Transparent adaptation: The channel set $C_{C\to S}$ is plit up into C_{hidden} and C_{trans}, C_{inf} is an additional output to the user making some inputs in $C_{C\to S}$ transparent to the user.

3.2.1.3. Diverted adaptation

Transparent adaptations are most desirable: users are able to understand adaptations if. However, complete transparency is illusory. Non-transparent adaptations are less desirable but sometimes sufficient if designers manage that adaptations work as expected.

There exists yet a third class of adaptations. Users may want to control the system's adaptations (e.g., because they are dissatisfied with the adaptations) but lack the proper (direct) input channel (e.g., for calibration [Fahrmair, 2005]). Instead they need to use an indirect channel that designers have not intended for user inputs.

This situation can be problematic. Users build up a mental model that allows them to infer relations between changes in the environment and the according system behavior. By intentionally influencing the environment to force the system into a certain behavior they exploit this mental model. However, the inferred relations easily miss the reality – i.e., the users mental model of the system's behavior and the real system behavior disperse. As a consequence, it is likely that their attempts to indirectly influence the system eventually fail.

The system fails its mission to relieve the users from interactions if users need indirect channels to force a system into a behavior. Causes are a wrong interpretation of contexts, wrong adaptation logic, inappropriate context model or insufficient user interface. If the system's adaptations are appropriate the users miss no input facility.

If the described divergence between expected adaptations and real adaptations results from changed user needs, changed business requirements, etc., the system needs modifications (or calibrations [Fahrmair, 2005]). However, the original requirements specification could already include the divergence due to imprecise or wrong elicitation of requirements. Avoiding these divergences is a strong argument for making

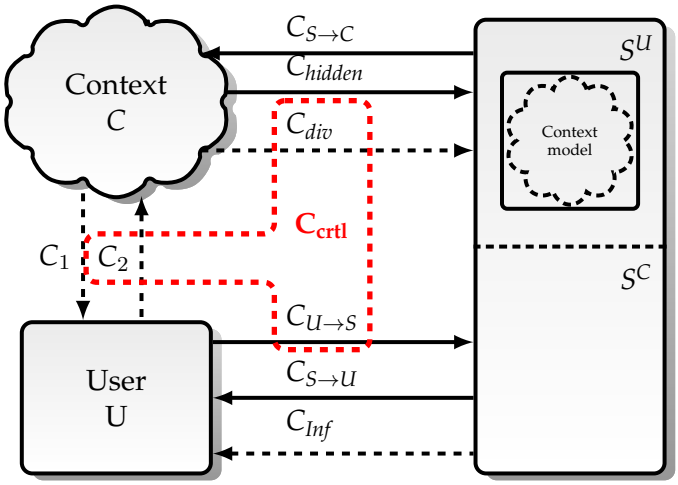


Figure 3.7.: Diverted adaptation: The user intentionally uses his channels to the context to force the system into a desired behavior instead of being able to use a dedicated channel to the system

assumptions and conclusions (i.e., the adaptation logic) explicit in the system specification enabling their validation and analysis.

Figure 3.7 illustrates diverted adaptations. The channel set C_2 models the dedicated manipulation of the environment. We separate the channels of the context input interface into channels $C_{hidden} = C_{C\to S} \backslash C_2$ which the system still uses for adaptations from the users point of view and channels $C_{div} = C_{C\to S} \cap (C_2)$ which the user misuses to control the system via C_2. The channel set $C_{crtl} = C_{U\to S} \cup C_{div}$ comprises all channels that the user controls directly.

DEFINITION 19 (DIVERTED ADAPTATION [BROY ET AL., 2009]):
Let be C_{div} and C_{crtl} as defined above. We call an adaptation diverted iff:

$$\{(i_1, i_2) \mid i_1, i_2 \in dom(S) \wedge i_1{\downarrow_{C_{crtl}}} = i_2{\downarrow_{C_{crtl}}} \wedge (S.i_1){\downarrow_{C_{crtl}}} \neq (S.i_2){\downarrow_{C_{crtl}}}\} \subset$$
$$\{(i_1, i_2) \mid i_1, i_2 \in dom(S) \wedge i_1{\downarrow_{C_{U\to S}}} = i_2{\downarrow_{C_{U\to S}}} \wedge (S.i_1){\downarrow_{C_{S\to U}}} \neq (S.i_2){\downarrow_{C_{S\to U}}}\}$$

With the use of the additional inputs the system appears more deterministic to the user. □

Usually there is always an interaction between the user and the environment. This interaction is intended even if it influences the system. After all, this is the goal of context-adaptive systems: observing users and their interaction with the environment and adapting their behavior accordingly.

The definition uses a special channel set C_2 that users may abuse. Detecting the abuse of a channel by the system during runtime is impossible. Instead, system engineers have to analyze interactions between the users and the system and match them to

the adaptations the system offers. Identifying the abuse of indirect channels at design time includes analyzing similar systems and exercising a decent requirements engineering. Both support a careful elicitation of the context model and the adaptation logic.

EXAMPLE 3.6 (DIVERTED ADAPTATION)
Again the navigation system changes the route because of a traffic jam. Furthermore, we assume that the user wants to use the road with the construction because she has further knowledge about the construction or has some errands nearby. The considered car navigation lacks an input possibility for the user to give instructions to avoid the adaptation of the route. Instead, the user may unplug the antenna for the Traffic Message Channel (TMC) thereby hindering the system from recognizing the construction (actually blocking a contextual input). ♣

3.2.2. System dimension

In contrast to the user dimension, the system dimension focuses on the structure of specifications. The structure of formal specifications should not influence the system architecture. But if the structure of the specification is appropriate, it supports the specification's understanding, validation, and finally its verification. According to the definition of stream processing functions in Section 2.4, it is possible to structure the system specification into a function with the signature $\mathbb{H}(C_{U\to S} \cup C_{C\to S}) \to \mathcal{P}(\mathbb{H}(O_s))$.

3.2.2.1. Ingrained adaptations

Systems classified as *ingrained adaptive* deeply integrate the processing of contextual information into their calculations. The specification has no additional structure representing the separated use of contextual information and user-provided information. As a consequence, the specification has the signature

$$\mathbb{H}(C_{U\to S} \cup C_{C\to S}) \to \mathcal{P}(\mathbb{H}(O_s)).$$

System engineers may have selected inputs for generating a context model and the elicitation of requirements but this separation is not visible in the structure of the specification. This is most applicable for functions whose contextual inputs are subject to frequent updates with a continuous nature.

From the viewpoint of writing and structuring specifications the class of ingrained adaptations is of little interest. However, there are other aspects of context-adaptive systems like, e.g., acquisition, distribution and update of context information – especially in scenarios with multiple integrated and interacting context-adaptive systems – that are independent of the structure of the specification. In such scenarios classifying systems as context-adaptive (even if ingrained adaptive) opens up for a selective treatment of certain information but without enhancing the behavioral specification.

EXAMPLE 3.7 (INGRAINED ADAPTATIONS)
Take the example of the adaptive steering (AS) in a modern car. The steering angle depends on the angle of the steering wheel (user input, here x) and on the current speed (context, here y). We examine a simplified and hypothetical function AS : $[-360;\ +360] \times [0;\ 250] \rightarrow [-45;\ 45]$; $f(x,y) \mapsto (1-(0,0036y)) * x$. *The resulting steering angle is z.*

According to former definitions we consider this function as context-adaptive. However, the contextual nature of y has no influence on structuring the function. On the other hand, it may happen that the speed is subject to uncertainty (e.g., there are multiple sources for the speed of a car sometimes providing different values) or needs to be updated in a certain way. In this case an explicit context model supports the (technical) handling of the speed information. ♣

3.2.2.2. Parametric adaptations

The idea of parametric adaptations applies if some contexts are of a discrete nature and their change frequency is low. We achieve this, e.g., by mapping received – possibly continuous data – data into a discrete set of equivalence classes which we identify with contexts. We denote this set as *SIT*. Note that *SIT* can be a Cartesian product that captures multiple aspects of a context. Using *SIT*, we represent the structure of parametric adaptive specifications as functions

$$[\mathbb{H}(C_{U\rightarrow S}) \times (\mathbb{H}(C_{C\rightarrow S}) \rightarrow SIT)] \rightarrow \mathcal{P}(\mathbb{H}(O_s))$$

Recall that we include finite I/O-histories in our considerations allowing to map finite behaviors to phases in the complete I/O-history. First, the function evaluates the contextual inputs and maps them into the set of contexts *SIT*. This step corresponds to a classification of contextual inputs and matches the idea of a context as an equivalence class of received contextual information. The function then uses the value of this mapping in the system function as a parameter. For a phase of execution this value is arbitrary but fixed.

EXAMPLE 3.8 (PARAMETRIC ADAPTATIONS)
We regard the adaptive steering in a modified version: Let x be the steering angle and y be the angle of the steering wheel. The requirements demand a relation between steering angle and the angle of the steering wheel as follows:

- *for speeds* $[0;\ 5[$ *the relation is* $y = 2 * x$
- *for speeds* $[5;\ 60[$ *the relation is* $y = x$
- *for speeds* $[60;\ \infty[$ *the relation is* $y = 0.8 * x$

We define the function eq : $\mathbb{R}_0^+ \rightarrow \{0.8, 1, 2\}$ *as:*

$$eq(n) = \begin{cases} 2 & \Leftrightarrow 0 \leq n < 5 \\ 1 & \Leftrightarrow 5 \leq n < 60 \\ 0.8 & \Leftrightarrow \textit{else}. \end{cases}$$

The function that calculates the resulting steering angle is then defined as:

$$y = p * x$$

with $p = eq(Current\_speed)$ ♣

3.2.2.3. Functional adaptations

By parameterization the function decouples the actual calculations of the usage functions and the handling of context information. However, the basic algorithms for the calculations of the usage functions stay identical.

Context-adaptive systems often offer different functionality in different contexts. A comprehensive way to structure specifications with respect to relations between contexts and different usage functions are specifications with the signature

$$\mathbb{H}(C_{C\to S}) \to [\mathbb{H}(C_{U\to S}) \to \mathcal{P}(\mathbb{H}(O_s))].$$

Again, we can apply a decoupled classification of contexts into equivalence classes with respect to the desired behavior:

$$[\mathbb{H}(C_{C\to S}) \to SIT] \to [\mathbb{H}(C_{U\to S}) \to \mathcal{P}(\mathbb{H}(O_s))].$$

Parametric adaptation is a sub-class of functional adaptations. As a difference parametric-adaptive systems need no complex mechanism (adaptation logic) to choose a functionality.

EXAMPLE 3.9 (FUNCTIONAL ADAPTATION)
We can easily structure the specification of an adaptive cruise control of a car as a functional adaptation. The vehicle controls the speed by different algorithms depending on the current driving situation. With a heading car, it controls the speed according to the distance to the heading car and the current speed. With a free lane, the car controls the speed with respect to a preset maximum speed. The two algorithms are different and even use different inputs for determining their outputs to the engine and brakes. The system chooses the effective algorithm at each time by a dedicated algorithm that evaluates the contextual inputs (here the distance to a leading car). ♣

3.2.3. Flexibility dimension

A main concern of context-adaptive systems is their ability to derive the current needs of the users. We assume that no system can change its overall behavior – the system's code determines its overall behavior. However, the algorithms to identify contexts and to choose a current behavior with respect to identified contexts can be arbitrarily complex.

Integrating more or less sophisticated algorithms allows a system to infer user needs, to choose from a set of known functions, or even to recombine functions to provide adequate functionality. In this section we classify context-adaptive system specifications with respect to the flexibility they offer. The flexibility ranges from a fixed relation between contexts and usage functions to learning algorithms which defer the definitive decision about mappings from contexts to behaviors until execution time.

3.2.3.1. Predetermined adaptation

Predetermined adaptation defines a set of usage functions and relates them with a set of contexts. This is the least flexible choice because the adaptation logic cannot change at run-time. The relation between usage functions and contexts is fixed. On the other hand, we can analyze this variant in advance.

Applications that are, e.g., mission/safety critical in the automotive- or avionic domain need to satisfy properties of functional safety. Their validation and verification ensures the appropriateness of their behavior. In such cases, context adaptation reduces to a structural concept for organizing specifications supporting the separation of concerns: specifying system behaviors in certain contexts is separated from concerns of relating contexts and behaviors.

Specifications with predetermined adaptations need a considerably different design process. The requirements elicitation and requirements specification phase are extensive finally capturing as many use-cases and contexts as possible and extracting the context model out of them [Yu, 1997; Sitou and Spanfelner, 2007; Sutcliffe et al., 2006, 2005]. The complexity of the systems is particularly caused by the number of use-cases, related functions, and interactions between functions. This requires a sophisticated technique to write and analyze system specifications.

3.2.3.2. Resource-triggered adaptation

Resource-triggered adaptation describes a system's ability to change the realization of functionality dynamically and to add new functionality at run-time. Up to now, we argued that a specification abstracts from implementation details like the architecture. However, the goal of some context-adaptive systems is working in settings with varying availability of entities that provide resources or functionality. Such settings often include mobile devices. Therefore, the dynamic restructuring of parts of the architecture and the ability to add new functions is part of the overall functionality of the system and hence one needs to specify it. This is a reasonable exception to our claim for

specifications being bare of architectural details.

Especially the web-service community discusses such settings [Aksit and Choukair, 2003; Booth et al., 2004; Mohyeldin et al., 2005; Berardi et al., 2005; Wu et al., 2003]. [3] A common use-case for such settings include users who request a certain function. The function is a (web) service. The web service can be complex and include a number of tasks which vary depending on the actual request. Tasks can be delegated to external (web) services. For this purpose, services provide information about functionality they export and functionality they import (require). This allows a flexible combination of available services at run-time. The actual orchestration depends on available resources and possibly user preferences like maximum costs, etc.

To this end, the setting is close to the predetermined adaptation except for dealing with a (logical) architecture at the level of specifications. However, efforts are undertaken to advance the flexibility of service orchestration. The system replaces unavailable requested services dynamically by combining available services. New functionality registers with the system and is available, even if this functionality was not planned by the system designers at design time. The challenges in this setting are service discovery and service orchestration and making the result available to the users [Srivastava and Koehler, 2003].

Most notably, neither the user requested service nor the available services need to be predefined at system design time. Appropriate description techniques allow adding them at run-time. Of course, the description techniques need to be machine readable. We regard this as altering parts of the system specification at run-time [Fahrmair et al., 2006a]. A dedicated part of the system that realizes an orchestration algorithm realizes the altered parts of the specification automatically with respect to available implementations.

In general it is undecidable if a web service realization matches with a given web service specification without further knowledge or restrictions with respect to the functions [Richardson, 1968]. Therefore, most of the approaches use ontologies [Sirin et al., 2003; Berardi et al., 2005] like DAML-S [Connolly et al., 2001], WSDL [Chinnici et al., 2004] and OWL-S [Coalition, 2004] and use planning algorithms (e.g. [Wu et al., 2003]) for matching requested and provided functionality.

The dynamic characteristics of these approaches prevents an a priori analysis of the specification. Sometimes, the result of a service orchestration will behave like expected by the user, because the matching of requested service and provided service is sufficiently precise. However, a danger exists that the composition misbehaves or no adequate service providers are available.

[3] Note that the term *web service* describes two different concepts. On the one hand, it describes a (required) behavior. In this context it has characteristics of a specification. A dedicated part of the system (a kind of interpreter) uses the description to build a network of interacting entities which we exclude from our definition of the term service. On the other hand, the term web service describes the entity that realizes the behavior. In this context it has the characteristics of a component. However, in contrast to the classical notion of components, the architecture that links the web services is flexible. For a detailed discussion of the term web service and its relation to the term service as we use it, see, e.g., [Meisinger and Rittmann, 2008].

3.2.3.3. Learning

Machine learning is one of the earliest attempts to enable systems to act in unknown environments. Enhancing context-adaptive systems with machine learning allows systems to learn about the relation between contexts and functionality. This has two aspects: I) classifying contexts with respect to contextual inputs and II) choosing the functionality (out of a set of available functions, parameters, etc.) that the system provides to the users in identified contexts.

Utilizing machine learning moves the complexity of analyzing and modeling scenarios at design time to the specification of adequate cost- and input evaluation functions to make sure that the system learns as intended. In [Alpaydin, 2004] Alpaydin defines machine learning as optimizing parameters of a pre-determined algorithm with respect to a model of the considered problem. The success of machine learning depends on a proper choice of such a model, the parameters, cost functions, and algorithms to update and process knowledge [Dennett, 1984]. This choice can be similar complex and error prone as the faithful specification of the behavior. A prominent example is the failure of single layer perceptrons to learn functions that are not linearly separable [Minsky and Papert, 1988].

In the context of learning, a specification describes an aspired effect instead of a specific learning approach. We illustrate this with an example: a specification better describes the information that a proper implementation is able to stack up two boxes instead of referring to a certain learning algorithm, cost function, etc. It is possible to implement this specification even without learning [4].

It is illusory to regard every possible user during specification time to provide functionality that is appropriate for all individual user demands. Such a requirement strongly suggests the use of machine learning to get closer to satisfying this requirement over time. Of course this requires a good choice for a learning algorithm, cost function, and observed aspects to allow a proper deduction of the parameters that the learning algorithm tweaks. Leaving open the relations between contextual inputs, situations, and provided functions in the specification with the goal of learning those gives more flexibility to the system while the predetermined adaptation fixes the possible adaptations once and for ever.

As a drawback, analyzing such specifications in advance is hard or even impossible. The compliance of the implementation to the specification is not guaranteed and important properties are not proven.

3.3. Summary

Context adaptation affects a broad range of aspects. In this section we discussed aspects that are relevant for this thesis. We ignored any other aspects about architectures, frameworks, data pre-processing, sensor fusion, etc. In this thesis we focus on a specific

[4] For some tasks it is too complex to explicitly regard every possible case in the code. Therefore, learning seems to be an adequate alternative then.

subset of context-adaptive systems with respect to the given classifications. Table 3.1 summarizes this subset.

Concerning the user dimension, the goal is to achieve transparent adaptation or at least a sophisticated variant of non-transparent adaptation. Diverted adaptations are not expedient and shall be avoided. We support this by providing a language for creating structured specifications of context-adaptive systems. We make the adaptation logic explicit and enable the validation of the appropriateness of adaptations and the analysis of the specifications for inconsistencies and incompleteness.

With respect to the functional dimension we focus on parametric and functional adaptations. We explore models and methods that allow decomposing specifications into functions for selecting usage functions and the usage functions themselves. The goal is a separated analysis of the resulting parts. Formally, we assume a finite, reasonable small set of contexts such that it is possible to define separate behaviors for them.

Properties like functional safety and reliability are important for embedded systems in domains like automotive and avionics. System engineers need to consider such properties in all stages of a system's life-cycle and especially during the development. The analysis of specifications for contradictions is an important aspect of this consideration. Therefore, we focus on predetermined adaptations.

	ingrained	parametric	functional
predetermined	-	X	X
resource-triggered	-	-	-
learning	-	-	-

Table 3.1.: Scope of the thesis with respect to the taxonomy

The definitions hide the actual efforts for creating the user model. Defining the user model includes analyzing the use of the system and the relevant contexts of this use. Defining and separating "relevant" contexts is a matter of defining the system idea and restricts the possible uses of the system. This consideration is especially important for the requirements elicitation phase. Thoroughly analyzing the users, their usage of the system, and the functionality to help the users, is necessary.

3.4. Related work

The idea of systems being aware of their operational environment and taking this information into account for their calculations has been a vision ever since computers became available to a broad community. Related ideas that base on the principles of context-awareness and -adaptations are, e.g.,

- *Ubiquitous computing, ambient intelligence, pervasive computing* with interacting agents supporting users with their tasks while being an integral element of the environment – often hidden from direct perception of the user [Weiser, 1991].

- *Autonomic computing* with the self-x (x $\in$ {healing, configuring, optimizing, protecting}) capabilities to reduce administrative efforts [Kephart and Chess, 2003].

- *Organic computing* with similar goals as autonomic computing but with a strong focus on using principles of nature to invent algorithms [Müller-Schloer, 2004].

Subsequently, we discuss related work including definitions, related terms and concepts that are close to the idea of context-adaptive systems. To this end, we omit work that is directly related to the behavioral specification of context-adaptive systems and refer to the end of Chapter 4.8 to allow a delimitation and comparison of this thesis' approach and other approaches.

3.4.1. Definitions

We tailor our definitions of the terms "context-adaptive systems", "context" and "situation" for the development of systems, especially for specifying their behavioral aspects. A number of publications present architectures that support a decoupled acquisition, interpretation, and distribution of context information. Their definitions differ from ours.

Schilit et al. defined context-adaptive systems to be able to adapt their behavior according to a user's position and nearby people, hosts, and accessible devices [Schilit et al., 1994]. This definition is strongly influenced by the *parc tab* applications and the *active badges* [Want et al., 1992]. Both offer services based on information about a user's location in an office environment. Ryan et al. define context as sensed information about a user's environment and gives examples like location, time, temperature, and ID. In the definition of Brown [Brown, 1996] context are elements of the users' environment which the computer knows about and enumerates, e.g., location, adjacency of other objects, imaginary companions, and time.

All of these definitions are example based. Newer definitions are more general. Dey et al. present work [Dey, 1998, 2000] that defines context-adaptive systems by using information that enhances the user's experience. Again they augment the definitions with enumerations of relevant aspects of context. Dey introduces a context server as a central entity for acquiring and processing context information (e.g., data mining, data fusion, interpretation, etc.) and postulates that the information stored on this server is the context. [Biegel and Cahill, 2004] gives a similar definition: he defines context as any additional information from the environment that describes a (sentient) object. Both definitions provide an intuitive understanding of the term context-adaptive system. The methodological support is restricted to the handling of the context information in distributed systems i.e. collecting information, interpreting it and making it available to applications.

The definition that Winograd gives is more specific: he defines context as additional information guiding the interpretation of the inputs of a user [Winograd, 2001]. This definition is close to the definition of the context model in Section 3.1.2 but yet informal.

Crowley et al. [Crowley et al., 2002] present a formal definition of context. They define states which they represent as predicates. Each state corresponds to a node in a graph. Arcs have labels with actions and connect the nodes. A user has goals that correspond to some of the nodes in the graph. A task is a relation between a current state, a goal state, and a possible path from the current state to the goal state. The context is the different roles that entities (other systems, people and the user herself) may take on the path from the current state to the goal state and the respective roles relations. Crowley et al. avoid listing selected information as context. In the context of this thesis we may use the information about the tasks in a specification to define the change of offered functions.

In a follow-up publication Coutaz et al. define context as a part of a process of interactions with a changing environment [Coutaz et al., 2005]. This definition regards user goals and their motivation by a history of interactions with systems. It supports elaborating a context model and defining rules that control the systems functions accordingly. In the thesis at hand, we focus on specifying such rules i.e. we take the results of the activities proposed by Coutaz et al. as an input to the formalization of the system specification.

McCarthy et al. offer a logic based model of context [McCarthy and Buvac, 1998]. The approach formalizes contexts with the goal of axiomatizing and reasoning in different domains. In this approach the truth value of statements depends on a current context. The use of context is for relating knowledge in different contexts and not for guarding usage functions. The approach supports the elaboration, analysis, and validation of a context model that defines guards for requirements and may serves as an input to the approach of this thesis as well.

Taking a closer look at the aforementioned definitions we notice that the specific goals of an approach influence the definition of "context" and related terms. This is as expected and complies with our claim of a system's ability to adapt being a matter of the viewpoint. It is rather a methodological aspect to treat inputs as context than a rigorous rule. The definitions in the thesis at hand are no exception. They rely on classifying certain inputs as context but leave open the choice. Depending on an application domain system engineers may choose different user models and related interaction possibilities. Given the choice of a user model and the related channels the definitions offer means to distinguish between direct user inputs (intended interaction) and indirect user inputs. Fixing a context eases the reasoning about consequences of context changes and the misuse of indirect channels.

3.4.2. Representation and handling of context

In addition to definitions of the term "context" there are a number of proposals for context representation. Most approaches in the literature use an attribute-value model [Baldauf et al., 2007]. These attribute-value models are suitable for specification purposes because they easily map to data structures and functions on these data structures. Other approaches utilize ontologies [Da and Zhang, 2004; Chen et al., 2003]. Ontologies allow to aggregate similar concepts. In a distributed environment with lots of devices and ap-

plications designed by different vendors this is one means to achieve a harmonization of terms. For the sake of specifying single applications this is less important.

The definitions of context-adaptive systems and context often accompany the presentation of architectures and frameworks. Architectures and frameworks deal with I) questions about the source, interpretation, storage and consistent distribution of contextual information and II) ensuring proper communication between applications even if designed by different vendors. Most approaches use layered architectures (e.g., [Dey et al., 2001; Fahrmair, 2005; Chen, 2004; Gu et al., 2005] including a sensor layer, a layer for data preparation, and a storage layer whose purpose is to provide the pre-processed data to any application built on top. The frameworks intend to integrate a number of applications while ensuring the common use of consistent data.

The context model as defined in Section 3.1.2 provides the interface of an application under development and the actual infrastructure (e.g., a central context server realizing the described layers). The context-model defines the information the system requires for its adaptations. Figure 3.8 illustrates the situation.

The context model allows to reason about the matching of an application under development into an existing infrastructure. We exclude the actual acquisition and interpretation of information from our considerations. Our abstract model rather deals with logical information captured in the context model. In a nutshell: literature about frameworks focuses on the acquisition, processing, and distribution of information. We focus on the usage of the already processed information.

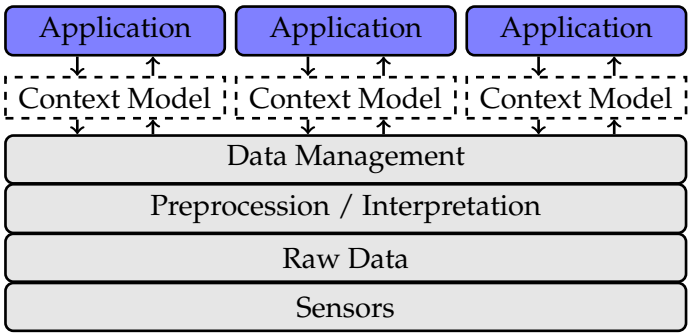


Figure 3.8.: Layered architecture of context-adaptive systems with the context model as the interface between frameworks and applications.

The idea of decomposing an application into a core system and an adaptation subsystem as presented in Section 3.1.1 goes back to ideas originally presented in [Fahrmair, 2005; Fahrmair et al., 2006a] and were formalized in [Broy et al., 2009]. Other approaches present similar concepts for the separations of concerns of the core system and the adaptation sub-system. Güdemann et al. [Gudemann et al., 2008], e.g., propose to decompose organic computing systems into an observer/controller layer and a functional layer. These layers roughly correspond to the adaptation subsystem and the system core respectively. Trapp lists the need for a distinct modeling of the adaptation and reconfiguration functions as a main motivation for his thesis. He justifies the need with the complexity of understanding adaptations in complex systems [Trapp, 2005].

Chapter 4

Modeling Functions (the MARLIN specification language)

The discussion in Chapter 3 shows that context-adaptive systems are complex systems. Main reasons are the number of usage functions, their interrelations, and the additional function that controls the availability of usage functions. Structuring these systems helps dealing with the complexity. Furthermore, making the three aspects explicit in a specification enables their analysis.

In this chapter we present the MARLIN specification language. It is applicable to functional specifications of context-adaptive systems and focuses on the aforementioned three aspects. MARLIN describes functions as services that a system offers. Services are user observable behavior and can be combined with other services to build more complex services.

We start the chapter with an introduction to service based specifications in Section 4.1 and continue with defining atomic services in Section 4.3. Subsequently, we present the composition operators. We distinguish between simple service composition (presented in Section 4.4) – which is appropriate to specify the core system – and conditional service compositions (presented in Section 4.5) – which is applicable for capturing the adaptation logic and to establish modes. The special operators in Section 4.6 allow transferring a service based specification into a system specification. Finally, Section 4.8 presents related work.

Contents

4.1. A short introduction to services

Services describe the interaction with the system without anticipating an architecture. This supports the understanding of the functional concepts of a system before developers make first design decisions. Services are at a higher level of abstraction then component based approaches.

The nature of services is different from the nature of components. Before we go into details with the MARLIN language, we introduce the basic concepts of service based specifications, and point out the differences to components. We argue for their appropriateness to give abstract specifications of behavior and discuss some peculiarities of MARLIN as a service based specification language tailored for context-adaptive systems.

4.1.1. Services and components

Use-cases are common to describe the behavior of a system under construction in the requirements engineering phase [Cockburn, 2000; Bruegge and Dutoit, 2003]. Use-cases are written down, e.g., as structured text or diagrams like message sequence charts [Krüger et al., 1999]. Each use-case represents a particular aspect of interacting with

the system. Taken together, a set of use-cases is an (often informal) specification of the system's behavior.

Sitou [Sitou, 2009; Sitou and Spanfelner, 2007] applies this idea to context-adaptive systems in a contextual requirements engineering approach. He combines use-case based descriptions of system behavior with the contexts the behavior is observable in, into contextual requirements chunks (CRCs). Each contextual requirement chunk is a partial description that contributes to the system's specification with respect to a part of the interface and the considered contexts.

Services are much like use-cases [Broy et al., 2007]. They formalize the possible interactions with the system while being a partial function along the primitives introduced in Chapter 2. The benefit of using services as formalizations of requirements is a direct mapping into a formal framework and the opportunity to analyze the requirements for conflicts and contradictions. To emphasize the differences between services and components we shortly compare them.

Services Services describe parts of the observable behavior. They formalize the functional requirements and capture only functional aspects. This allows focusing on the functional correctness and appropriateness of the concepts of the system. A service is a clipping of the behavior of a system that is underspecified concerning internal structure and – making assumptions about the environment – supports the definition of partial behavior [Schätz and Salzmann, 2003]. The interaction with the system and the interrelation of offered functionality are the main concerns of service based specifications. A service based specification describes no aspects of possible solutions. We say that services describe the problem domain [Harhurin, 2010].

Components In contrast, component-based approaches use components as reusable units of behavior and structure [Schätz et al., 2003; Stølen, 1996]. Components are a means of designing an architecture and already contribute to the solution domain. Modularity, information hiding and concurrency are the aspects that drive most component based approaches. Information hiding, a strict notion of interface, and interface contracts enable a modular reasoning about parts of a system and a separated development and maintenance of the constituents. This principle becomes obvious in design techniques like design by contract [Meyer, 1988] and assumptions/guaranties [Abadi and Lamport, 1995]. Note that architecture descriptions are specifications as well but exist at a lower level of abstraction where first design decisions were already made. We say that components describe parts of the solution domain [Harhurin, 2010].

While component driven approaches focus on compositionality, services put their focus on overlapping views. Compositionality is necessary to allow replacing some entity (possibly a component) by another one with the same observable behavior but having a different structure or implementation internally. This is of no use in the context of services. Since a service shall define no structural obligations for any implementation, it is no considerable scenario to replace a service by another one that behaves

equally but has a different structure or implementation. The only reason to replace a service is if the new one has a different behavior or affects different resources. Table 4.1 summarizes the most important differences.

Services	Functions that describe interactions with the system at its interface; possibly describing different viewpoints of stakeholders	Can be overlapping with respect to input channels, output channels and local data which can cause conflicts; modularity and compositionality are no concern
Components	Interacting modules that contribute to providing a service; Components may be internal and not be observable directly by a system user	Capsule local data and implementation details to enable separation of concerns and refactoring. Components cannot conflict but it is possible that their interfaces do not match; Components shall be modular and compositional to allow replacing an implementation by another one if the observable behavior remains unchanged

Table 4.1.: Summary of properties of services and components

A strong relation exists between services and components.

- It is possible to use services to describe the interface behavior of a component.
- A network of communicating components can realize a component of a higher level in an architectural hierarchy.
- A network of communicating components can realize a service making it an offered service of the component(s).

We discuss the relation between services and components and its methodological use in more detail in Section 5.5.

Service description techniques like, e.g., [Schätz and Salzmann, 2003; Broy, 2005] and [Botaschanjan et al., 2008] compose specifications out of concurrent services. Services correspond to horizontal projections on the systems' behavior with respect to a clipping of the system's interface. The JANUS approach introduces the horizontal projections as so-called *slices* [Broy, 2005]. Figure 4.1 illustrates such slices. Note that sub-services may comprise overlapping sets of I/O channels. This causes services to interfere or even to conflict.

In an ideal world, a system engineer knows about the behavior of a system while designing the architecture: the functional specification carries this information. The

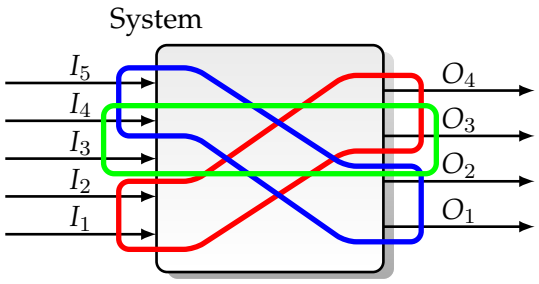


Figure 4.1.: Projections on the system interface capturing the I/O-relations of services

challenge for the designers of the architecture is the mapping of the behavior to a set of components such that the composition of the components satisfies the functional specification i.e. the architecture refines the functional specification (see Figure 1.1). This includes deducing component requirements from the system requirements as presented, e.g., in [Penzenstadler, 2011]. Non-functional requirements (like, e.g., structure of the organization, patterns, etc. [Broy et al., 2008; Penzenstadler, 2011]) influence the architecture.

In contrast, one usually has only incomplete and often conflicting information about the intended system behavior while creating the functional specification. If the information was complete and consistent the work of composing a specification out of partial behavior descriptions would already be finished. Therefore, a main challenge is finding a means for describing parts of the functionality of the system that match the already given information and composing them into a concise system specification.

Finally, system engineers are interested in a consistent specification to proceed with the architecture work in the next development phase. Nevertheless, the ability of modeling conflicts is an important property of services at the beginning of the specification work. The engineers formalize the requirements, different (and possibly overlapping) views, etc., exactly as the stakeholders describe them without further processing. This ensures that the information from the stakeholders is correctly captured in a preliminary formal specification and not filtered by the engineers. Subsequently, the engineers analyze the preliminary specification for conflicts and unconsidered use-cases and discuss them with the stakeholders. Hence, the stakeholders can decide about the handling of existing deficiencies. Usually this results in a better result (user acceptance) then forcing the engineers to make these considerations.

4.1.2. Services in MARLIN

MARLIN is a service based specification language to map context-related descriptions of system behavior like the contextual requirements chunks of Sitou into a formal framework. For this purpose, MARLIN introduces additional projections. In addition to horizontal projections that decompose behaviors with respect to their interface, MARLIN allows temporal projections that decompose behaviors with respect to phases of execution. We call these projections horizontal.

We follow the definitions from Section 3.1.1 and distinguish between language aspects to specify the core system and those to capture the adaptation logic.

Core system The core system captures the interactions between the user and the system under construction that are valid for different contexts. According to the timely restricted nature of contexts, MARLIN provides means to describe behavior for an arbitrary, possibly limited duration of time with services. Parallel and alternative compositions allow a bottom-up composition of complex services out of simpler ones.

Adaptation subsystem Mode transition systems capture the change of behaviors. The nature of the composition by mode transition systems is sequential. Mode transition systems handle the aspects of changing the observable behavior like choosing a succeeding behavior, determining the exact time of change, and treating intermediate results of calculations. As a peculiarity, mode transition systems are useful for establishing system modes as well. Any inputs can trigger the change of system modes, not just contextual inputs.

Sequentially composed services are without conflicts. Between subsequent behaviors only an indirect influence is possible via the handover of intermediate results. This enables an independent reasoning for conflicts between services with respect to the modes they occur in [Maraninchi and Rémond, 1998].

4.2. The MARLIN specification language

Subsequently we present the syntax and semantic domain of MARLIN. The semantic mapping of the operators based on this semantic domain is the topic of the rest of this chapter.

4.2.1. Syntax of MARLIN

Service based specifications form a hierarchical structure. Atomic services are the leaves of this hierarchy. More complex services are composed of simpler ones. Atomic services are the only ones whose behavior has to be given explicitly. Any compound service's behavior follows from its sub-services' behaviors and the used composition. The root of the hierarchy is a service that represents the system under construction.

MARLIN specifies services by service expressions. We may name services to allow an easy embedding in other expressions. Note, embedding service expressions in multiple other expression is different from instantiation (which is possible in component approaches). Each occurrence of the embedded expression exactly refers to the same channels and therefore opposes the same restrictions to these channels.

We use a denotation similar to BNF [Knuth, 1964] to present the MARLIN language. We enclose nonterminals by pointed brackets $<>$. In addition we extend the BNF by the symbols $+$ and $*$ in their meanings of regular expressions as, e.g., introduced in

Section 2.3 to abbreviate repetitions. Grouping for the application of $+$ and $*$ is by rectangular brackets [] because they do not appear in the MARLIN language.

For the following definitions let $ID_S \subseteq NAMES$ be a set of service names. We will use $A, B, \ldots$ as well as $f, f_1, f_2 \ldots$ as representatives of these names. Let <S> denote the initial non-terminal for a service.

<S>	::=	<ID_S>	// service name
	|	FALSE	//denying service
	|	TRUE	// accepting service
	|	NOP$_{<V>}$	//no operation
	|	SAME$_{<V>}$	//no state change
	|	<AS>	// atomic service
	|	(<S>)	// parenthesis
	|	$\tau_{(<varset>)}$(<S>)	// variable hiding
	|	$\nu_{(<chset>)}$(<S>)	// channel hiding
	|	<S> $\otimes$ <S>	// parallel composition
	|	<S> $\oplus$ <S>	// alternative composition
	|	(<M>,<δ>,<Φ>)	// mode transition system

This is only the general language setup. We define the syntactic domains that are yet open in the respective paragraphs subsequently.

One service name takes the role of the root service. To assign behaviors to service names we give their respective service expression in a set of equations. Formally, a specification is a tuple (Ψ, N_1) with $N_1 \in (ID_S)$ as the root service and Ψ as a finite set of service expressions:

$$N_1 \stackrel{def}{=} F_1$$
$$\vdots$$
$$N_n \stackrel{def}{=} F_n$$

With names $N_1, \ldots, N_n \in ID_S$ being all the names occurring in any of the equations.

We say that a service expression is well formed if it is a correct word with respect to the grammar and if it is non-recursive i.e. names occurring on the left side of definitions never occur on the right side of the definition, neither directly nor transitively.

4.2.2. Semantic domain of MARLIN

In Chapter 2 we introduced I/O-histories as a semantic domain. This is useful for considerations about a system's behavior over its complete run-time. In MARLIN we focus on creating specifications by composing services. Services are partial and possibly cover the behavior for a phase of execution only (i.e., for a limited time interval) eventually being replaced by other services. The new service may use intermediate

results of the replaced service. Therefore, we need to introduce states capturing the intermediate results. This extends the semantic domain.

Let V be a set of typed variables. We introduce a valuation function

$$\sigma : V \to \mathbb{D}$$

that assigns a value from the universe of data to each variable from the set V such that $\forall v \in V : \sigma(v) \in TYPE(v)$. We call σ a state and write Σ_V for the set of all states over the set of variables V. If the set of variables is clear from the context, we drop the index to Σ.

To ensure a clean separation of the core system and the adaptation logic, we classify the variables into two sets. The set VC (called the **core variables**) contains all variables of the core system. The set VM (called **mode-variables**) contains the variables that mode transition systems introduce. We require both sets to be disjoint. The local variables of a service V are $V = VC \uplus VM$. The set VM is automatically maintained by the upcoming definition of mode transition systems.

DEFINITION 20 (SERVICE):
A service is a function that maps a state and a sequence of inputs to a set of sequences of outputs and states. For a syntactic interface $(I \rhd O)$ and a set of variables V we write $(I \rhd O, V)$ to denote the extended interface. The semantic functions is:

$$S : (\Sigma_V, \vec{I}) \to \mathcal{P}(\vec{O}, \Sigma_V) \tag{4.1}$$

Note that this is a big step semantics [Leroy, 2010]. We call I, O and V the resources of the service. Inputs I are observed resources. Outputs O and variables V are controlled resources. We adapt the set $\mathbb{I}$ of all interfaces and the sub-interface relation $\subseteq$ accordingly.

The states hand over information between subsequent services. The final state of one service becomes the initial state of its successor service determining the possible interactions starting in this state.

We capture the valuations of this set valued function in a tuple $(\sigma, i, o, \acute{\sigma})$ and call it a semantic tuple. We denote the set of all semantic tuples as $\mathbb{T}$. We write $[\![S]\!]$ to denote the set of all mappings allowed by a service expression S i.e. the semantics of this expression. □

We shall only accept those functions $(\Sigma_V, \vec{I}) \to \mathcal{P}(\vec{O}, \Sigma_V)$ as valid specifications for services that show three vital properties (Sections A.1 and A.2 discuss these properties and their impact in detail):

1. Restricted causality (based on the semantic domain)

$$\begin{aligned}&\forall i_1, i_2 \in \vec{I}, t \in \mathbb{N}, \sigma \in \Sigma : \\ &\qquad (i_1)_{\downarrow t} = (i_2)_{\downarrow t} \Rightarrow \\ &\qquad\qquad \{o \mid (o, \acute{\sigma}) \in F((i_1)_{\downarrow t+1}, \sigma)\} = \{o \mid (o, \acute{\sigma}) \in F((i_2)_{\downarrow t+1}, \sigma)\}\end{aligned}$$

This property is a general one that ensures the realizability of the system [Broy and Stølen, 2001]. However, the original idea of (strong) causality bases on infinite streams and input complete components. In the context of MARLIN behaviors are finite i.e. may refuse reactions to inputs. Hence, the original idea of strong causality does not work. We discuss this in detail in Section A.1 and illustrate it in Example A.1. Restricted causality is a necessary condition for strong causality and in fact restricted causality together with input completeness (i.e., removing all partiality) are necessary and sufficient conditions for strong causality.

2. Infix monotonicity:

$$\begin{aligned}&\forall(\sigma, i, o, \acute{\sigma}) \in [\![S]\!] : \#(i) > 2 \Rightarrow \\ &\quad \exists\, i_1, i_2 \in \vec{I_S};\ o_1, o_2 \in \vec{O_S}\, \exists\, \sigma_l \in \Sigma_S : \\ &\qquad i = i_1 \frown i_2 \wedge o = o_1 \frown o_2 \wedge (\sigma, i_1, o_1, \sigma_l) \in [\![S]\!] \wedge (\sigma_l, i_2, o_2, \acute{\sigma}) \in [\![S]\!]\end{aligned}$$

Property 2. ensures that any behavior can be interrupted at any time such that

a) an intermediate state exists

b) the first part stopping in the intermediate state is a valid behavior

c) the second part starting at the intermediate state is a valid behavior, and finally

d) interrupting and resuming a behavior will not introduce additional behaviors

This property is necessary to allow immediate reactions to changes in a system's context. Furthermore, it enables certain transformations in specifications (cf. Sections A.2 and A.3).

The shortest reasonable restricted causal behavior has two time intervals (one for receiving the inputs and one for communicating the outputs). Therefore, the shortest behaviors considered for the property have a minimum length of three time intervals.

3. Behavioral closure

$$\begin{aligned}&\forall(\sigma_1, i_1, o_1, \acute{\sigma_1}), (\sigma_2, i_2, o_2, \acute{\sigma_2}) \in [\![S]\!] : \\ &\quad \acute{\sigma_1} = \sigma_2 \Rightarrow (\sigma_1, i_1 \frown i_2, o_1 \frown o_2, \acute{\sigma_2}) \in [\![S]\!]\end{aligned}$$

Property 3. includes all possible extensions of behaviors to the set of valid behaviors.

The extended semantic domain includes variables. The variables define a state space. Valuations of the state space will serve as connection points of subsequent behaviors if different services are responsible for controlling the system's reactions in different phases of execution. In the context of the new semantic domain we extend the notion of restriction to variables.

DEFINITION 21 (RESTRICTION OF STATE SPACES AND SERVICE SLICING):
Let V and W be two sets of variables and $U = V \cap W$

$$\lfloor : \Sigma_V \times W \to \Sigma_W, \quad \sigma_V \lfloor_W \mapsto \sigma_U \quad \Leftrightarrow$$
$$\forall x \in U : \sigma_U(x) = \sigma_V(x)$$

According to the semantic domain for services we also adapt the notion of a slice:

$$S\dagger(I' \rhd O', V').(\sigma \lfloor_{V'}, x \lfloor_{I'}) = \{(y \lfloor_{O'}, \acute{\sigma} \lfloor_{V'}) \mid (y, \acute{\sigma}) = S.(\sigma, x)\}$$

We furthermore introduce the following abbreviation:

$$(\sigma, i, o, \acute{\sigma}) \lfloor_A \stackrel{def}{=} (\sigma \lfloor_{\Sigma_A}, i \lfloor_{I_A}, o \lfloor_{O_A}, \acute{\sigma} \lfloor_{\Sigma_A})$$

for some service A with interface $(I_A \rhd O_A, V_A)$ □

4.2.3. Discussion of the semantic domain

The semantic domain bases on the natural semantics presented by Kahn [Kahn, 1988]. The natural semantics replaces the stepwise reduction of sequential expressions by a single big reduction. Therefore, the semantics is also called a big step semantics [Leroy, 2010]. Similar to Kahn's natural semantics, the interactions in MARLIN's semantic start in an initial state and result in a final state at the end of the interaction. States model information that carries over between successive services. The initial state is an equivalence class for the interactions that happened before the activation of the service.

Internal states antagonize modularity in parallel compositions. However, to avoid states, we need to consider the complete event-history until the activation of a service. This is unreasonable if one wants to focus only on the possible interactions a service offers.

MARLIN derives longer lasting behaviors from shorter behaviors via the composition operators. Mode transition systems compose phases of execution into even longer phases that describe the interactions across changes of contexts. The initial and final states always mark the beginning and ending of such phases of execution.

Services maintain no states on their own. They just process states according to their behavioral definitions. The services use the states as a kind of input and output. They read the state once at the beginning of the phase and write it at the end of the phase. Possible intermediate states are hidden.

We strongly agree with the arguments and goals of information hiding for the modular development of systems. However, as a peculiarity of MARLIN – and similar to [Schätz, 2009] – we allow shared variables. We shortly argue for this choice and refer to Section 5.5 for an in-depth discussion of methodological implications and a useful integration into the development process.

- A specification is a black box description of the entities' behavior and contains no (or as little as possible) information about the realization of the entity. As a consequence, no information about the entities state is desirable. Therefore, the variable concept in MARLIN only eases, structures and finally facilitates concise and analyzable specifications. Distribution and modular development of the considered entities are out of scope. All details about the structure of the specification are removed at the final step of turning a service specification into a system specification as shown in Section 4.6.2.

- MARLIN is intended for the use in *interface specifications* of context-adaptive systems during the transition from an informal system description to a formal specification. We interpret this as formalizing use-cases. According to Broy [Broy, 2005] services are possibly overlapping *horizontal* projections on the behavior like illustrated in Figure 4.1. MARLIN complements the horizontal projections by temporal projections. If we accept variables as means I) to write concise definitions especially concerning the aspects of changing and resuming behaviors and II) to refer to common aspects of use-cases, we need to include variables into the overlapping projections. Regarding these intersections contributes to the modeling of shared aspects of use-cases and to revealing contradictions between them.

In a nutshell: the focus of the MARLIN specification language is the construction and analysis of specifications and not the modular development of architectures. The outcome of applying MARLIN is an interface specification that is a pure functional description without mandatory structure.

4.3. Modeling Atomic Services

In the sequel, we introduce atomic services and their semantics. Atomic services are the smallest entities of behavior. They represent a basic interaction with the system. A behavior is only specified for atomic services explicitly. Any other service's behavior, including that of mode transition systems, is derived from the atomic services' and the composition operators' semantics.

4.3.1. Abstract syntax of atomic services

An atomic service <AS> has a set of input channels I, a set of output channels O, and a set of variables V as well as a body B. Note that for any atomic service the set of mode variables VM is empty. Furthermore, any service needs to have at least one output channel, but can be without inputs or variables.

```
<AS>  ::=  in   {[<I>;]*},        //set of inputs
           out  {[<O>;]+},        // nonempty set of outputs
           var  {[<V>;]*},        //set of local variables
           spec <B>               //specification body
```

Let $ID \subseteq NAME$ be a set of identifiers ($ID \cap (ID_S \cup ID_M) = \varnothing$) and *TYPE* be the corresponding set of types. The input-, output channels and variables are defined as

```
<I>   ::=  <ID>:<TYPE>
<O>   ::=  <ID>:<TYPE>
<V>   ::=  <ID>:<TYPE>
```

The body *B* describes a set of single transition. We use the notation presented in [Huber et al., 1997] and [Grosu et al., 1996]. Subsequently, we explain this syntax.

```
<B>   ::=  {[(<PTN>);]+}
<PTN> ::=  {<PRE>} <INPUTPATTERN> / <OUTPUTPATTERN> {<POST>}
```

The patterns <PTN> consist of a precondition <PRE>, an input pattern <INPUTPATTERN>, a corresponding output pattern <OUTPUTPATTERN>, and a post condition <POST>.

The precondition <PRE> is a predicate over the local variables. Formally, it is a function

$$PRE : (V \to \mathbb{D}) \to \mathbb{B}.$$

Syntactically <PRE> is a string that satisfies the syntax of standard predicate logic. We omit the definition of this syntactic domain and refer to, e.g., [Church, 1996]. The transition is only enabled if the precondition is satisfied.

The input pattern <INPUTPATTERN> describes the inputs at the input interface. It is a condition on valuations of the input channels. Syntactically we write

```
<INPUTPATTERN> ::=  {<ID>?<PATTERN_EXP>}*
```

to indicate that the incoming message(s) at the respective channels satisfy the pattern <PATTERN_EXP>. By replacing all occurrences of ? by $=$ and conjoining them we transform the patterns into predicates. This turns the input pattern into a predicate

$$INPUTPATTERN : (I \to \mathbb{D}^*) \to \mathbb{B}$$

which returns TRUE if the received values within one time interval match the indicated values. Similar we define output patterns:

```
<OUTPUTPATTERN> ::=  {<ID>!<PATTERN_EXP>}*
```

to indicate that the service writes certain values to the output channels. Again the output pattern corresponds to a function

$$OUTPUTPATTERN : (((V \to \mathbb{D}) \times (I \to \mathbb{D}^*)) \to (O \to \mathbb{D}^*)) \to \mathbb{B}$$

that returns TRUE if the communicated values match the indicated values which are a function of the inputs and the state.

Recall that $\mathbb{D}^*$ is a finite untimed stream. A transition receives and writes finite untimed streams (c.f. Section 2.1). A communication history over time consists of finite untimed streams representing the messages communicated within one time interval.

The post condition <POST> fixes the values of the local variables after the transition depending on the previous state and the inputs. Formally it is a predicate

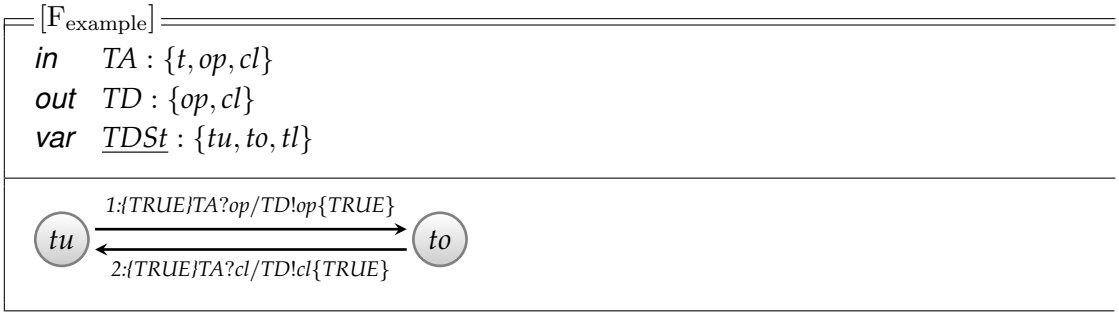
$$POST : (((V \to \mathbb{D}) \times (I \to \mathbb{D}^*)) \to (\acute{V} \to \mathbb{D})) \to \mathbb{B}$$

that returns TRUE if the output values are correct values according to the function that maps inputs and data valuations before the transition to data valuations after the transition. Thereby, $\acute{V}$ represents the values of the variables after executing the transition (similar to the notation used by Hehner in predicative programming [Hehner, 1984a,b]). Again the post condition syntactically is a string that matches the conventions for standard predicate logic.

4.3.2. Graphical syntax of atomic services

The abstract syntax is the basis for the semantic mapping but is bulky for specifying behaviors. Therefore, we present a concrete syntax based on the state transition diagrams (STDs) as presented in [Huber et al., 1997] and [Broy, 1997]. A STD is a labeled transition graph with nodes $\mathbb{K}$ which we call control states and arcs $\mathbb{V}$ with a labeling function $l : \mathbb{V} \to \mathbb{L}$ where $\mathbb{L}$ is the set of guards of type $\{pre\}inputpattern/outputpattern\{post\}$.

EXAMPLE 4.1 (STATE TRANSITION DIAGRAM)
As an example for a state transition diagram we use the trunk door position control. If the door is shut and unlocked while receiving a message `op` *on channel* `TA`*, the door opens. The state changes to* `to` *and a message* `op` *on channel* `TC` *causes the trunk door engine to open the trunk. If the door is open a message* `cl` *on channel* `TA` *closes the trunk door with the corresponding state change and output message.*



In the example we have

- *a set of nodes* $\mathbb{K} = \{tu, to\}$,
- *a set of arcs* $\mathbb{V} = \{(tu, to), (to, tu)\}$ *and*

- *two labels* $(tu, to) \mapsto 1$: *and* $(to, tu) \mapsto 2$:

where 1: *and* 2: *refer to the labels in the figure.* ♣

While control states are a good method to structure the representation of behaviors, they are no more than syntactic sugar and need no extra representation in the abstract syntax. We treat control states as additional conditions on a dedicated variable in the state space. In specifications we underline these variables to indicate their role in the graphical representation as control states. We transform a label

$(a, b) \mapsto \{pre\}in/out\{post\}$

in a specification that encodes the control state in a variable *pc* to a transition

$\{pre \wedge pc = a\}in/out\{post \wedge pc = b\}$

We omit pre- and post-conditions in labels that evaluate to *TRUE*.

EXAMPLE 4.2 (STATE TRANSITION DIAGRAM CONTD.)
The STD representation in Example 4.1 transfers to

$[\mathrm{F}_{\mathrm{example}}]$

in	*TA*
out	*TD*
var	$\underline{TDSt}$

$\{TDSt = tu\}TA?op/TD!op\{TDSt = to\}$
$\{TDSt = to\}TA?cl/TD!cl\{TDSt = tu\}$

♣

As an alternative, and if control states are less important we use a tabular style concrete syntax similar to [Broy, 2010]. The table has four main rows according to the four constituents of a transition.

pre			in			out			post		
v_1	...	v_l	i_1	...	i_m	o_1	...	o_n	v_1	...	v_l

This tabular representation is close to the abstract syntax and allows a more comprehensive overview over the valuations of the variables and channels.

4.3.3. Semantics of atomic services

In the sequel we will use the following abbreviations:

$$
\begin{array}{rcl}
I = & i_1;\ \ldots;\ i_l & //\text{names of input channels} \\
O = & o_1;\ \ldots;\ o_m & //\text{names of output channels} \\
V = & v_1;\ \ldots;\ v_n & //\text{names of variables} \\
I_F = & I_1;\ \ldots;\ I_l & //\text{types of input channels} \\
O_F = & O_1;\ \ldots;\ O_m & //\text{types of output channels} \\
V_F = & V_1;\ \ldots;\ V_n & //\text{types of variables} \\
I_F{}^{\underline{\omega}} = & I_1{}^{\underline{\omega}};\ \ldots;\ I_l{}^{\underline{\omega}} & //\text{timed streams over } I_F \\
O_F{}^{\underline{\omega}} = & O_1{}^{\underline{\omega}};\ \ldots;\ O_m{}^{\underline{\omega}} & //\text{timed streams over } O_F \\
\vec{I} = & i_1 \to I_1{}^{\underline{\omega}};\ \ldots;\ i_l \to I_l{}^{\underline{\omega}} & //\text{channel history over I} \\
\vec{O} = & o_1 \to O_1{}^{\underline{\omega}};\ \ldots;\ o_m \to O_m{}^{\underline{\omega}} & //\text{channel history over O} \\
\Sigma_F = & v_1 \to V_1;\ \ldots;\ v_n \to V_n & //\text{mappings of variables to values}
\end{array}
$$

DEFINITION 22 (SEMANTICS OF ATOMIC SERVICES):
Without limitation let F be an atomic service:

$[F]$

in $i_1 : I_1;\ \ldots;\ i_k : I_k;$
out $o_1 : O_1;\ \ldots;\ o_l : O_l;$
var $v_1 : V_1;\ \ldots;\ v_m : V_m;$

$\{pre_1\} inpattern_1 / outpattern_1 \{post_1\}$
$\ldots$
$\{pre_n\} inpattern_n / outpattern_n \{post_n\}$

The semantics of the service F is:

$$
\begin{array}{rcll}
[\![F]\!] \stackrel{def}{=} \mathcal{CL}.\{(\sigma, i, o, \acute{\sigma}) \mid \sigma \in \Sigma_F & \wedge & \textit{// value assignment for initial state} \\
i \in \vec{I} & \wedge & \textit{// assignment for the input channels} \\
o \in \vec{O} & \wedge & \textit{// assignment for output channels} \\
\acute{\sigma} \in \Sigma_F & \wedge & \textit{// value assignment for final state} \\
B & \} & \textit{// the specification body}
\end{array}
$$

Where B is the body of the specification. The body of the specification is the conjunction of the pre-, post- and I/O-conditions of the transition patterns:

$$B \stackrel{def}{=} \exists c \in \mathbb{H}(I \cup O) : i = c_{\lfloor I} \wedge o = c_{\lfloor O} \wedge$$

$$\bigvee_{i=1}^{n} pre_i(\sigma) \wedge inpattern_i(c_{\lfloor I}.1) \wedge outpattern_i(\sigma, c_{\lfloor I}.1, c_{\lfloor O}.2) \wedge post_i(\sigma, c_{\lfloor I}.1, \acute{\sigma}) \qquad \square$$

The list of transition patterns transfers to a disjunction of each respective condition. This bases on an idea of Pnueli and Manna [Manna and Pnueli, 1995] and is explained in more detail in [Clarke et al., 1999]. This representation allows combining multiple transitions of an automaton. We call $(\sigma, i, o, \acute{\sigma})$ a semantic tuple and denote the set of all semantic tuples as $\mathbb{T}$. Using the channel history c in the definition ensures that in the case of feedback values on output channels occur on input channels in the same time interval.

The function $\mathcal{CL}()$ is the transitive closure of the set. It generates sets of longer lasting behaviors (cf. big step semantics [Leroy, 2010]) out of the single steps that the basic service specification defines. This is necessary because we treat services as phases of execution that we use to describe the interaction with the system while a context is active (see later in Section 4.5). The closure is formally defined as:

$$\begin{aligned}
&\mathcal{CL} \in \mathcal{P}(\mathbb{T}) \to \mathcal{P}(\mathbb{T}) : \\
&\quad \mathcal{CL}.X \mapsto Y \text{ such that} \\
&\qquad Y = \mu Y.X \cup \{(\sigma, i_1 \frown i_2, o_1 \frown o_2, \acute{\sigma}) \mid \exists\, \sigma_l \in \Sigma, t \in \mathbb{N} \\
&\qquad\qquad (\sigma,\ i_1,\ o_1,\ \sigma_l) \in Y \wedge \\
&\qquad\qquad (\sigma_l,\ i_2,\ o_2,\ \acute{\sigma}) \in Y \}
\end{aligned}$$

We define

$$F.(\sigma, i) = \{(o, \acute{\sigma}) \mid (\sigma, i, o, \acute{\sigma}) \in [\![F]\!]\}$$

and adapt the notions of domain and range of services

$$\begin{aligned}
dom(F) &= \{(\sigma, i) \in (\Sigma, \vec{I}) \mid F.(\sigma, i) \neq \varnothing\} \\
ran(F) &= \{(o, \acute{\sigma}) \in F.(\sigma, i) \mid (\sigma, i) \in dom(F)\}
\end{aligned}$$

We introduce two dedicated atomic services with polymorphic I/O interfaces i.e. they adapt their interface to their context.

$$[\![FALSE]\!] \stackrel{def}{=} \{\} \tag{4.2}$$

$$[\![TRUE]\!] \stackrel{def}{=} \{(\sigma, i, o, \acute{\sigma}) \mid \sigma \in \Sigma \ \wedge \acute{\sigma} \ \in \Sigma \wedge i \in \vec{I} \wedge o \in \vec{O}\} \tag{4.3}$$

TRUE and *FALSE* need a polymorphic interface to allow neutrality and absorption in formulas with services that have different interfaces like: $F_1 \otimes (TRUE \oplus F_2)$. For the sub-expression $(TRUE \oplus F_2)$ we intend the result *TRUE* (absorption with respect to

alternative composition). Furthermore, for an expression $(F_1 \otimes TRUE)$ we intend the result F_1 (neutrality with respect to parallel composition). If F_1 and F_2 have different interfaces, *TRUE* must first adapt the interface of F_2 and then that of F_1.

Additionally we introduce two auxiliary services *NOP* and *SAME* that take care of local variables without affecting input or output channels. Their interface is fixed. Let V be a set of local variables.

$$[\![NOP_V]\!] \stackrel{def}{=} \{(\sigma, i, o, \acute{\sigma}) \mid \sigma, \acute{\sigma} \in \Sigma_V \wedge \sigma = \acute{\sigma} \wedge i = o = \langle\!\langle\rangle\!\rangle\} \tag{4.4}$$

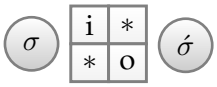
$$[\![SAME_V]\!] \stackrel{def}{=} \{(\sigma, i, o, \acute{\sigma}) \mid \sigma, \acute{\sigma} \in \Sigma_V \wedge \sigma = \acute{\sigma}\} \tag{4.5}$$

The service NOP_V is a "no operation" service that requires that the considered variables V remain unchanged. In addition, the I/O-streams have to be empty. In contrast, the service $SAME_V$ accepts any I/O-history of arbitrarily length but requires the variables to remain unchanged. Both are auxiliary services that help writing specifications.

4.3.4. Further explanations to the semantics

The semantic domain has some peculiarities that are mainly caused by restricted causality. Therefore, we discuss these peculiarities.

The simplest service is a single transition from one state to another, while reading inputs in one time interval and writing outputs in the next time interval. Any simple transition maps to a semantic tuple that captures the interaction with the environment for two consecutive time intervals:



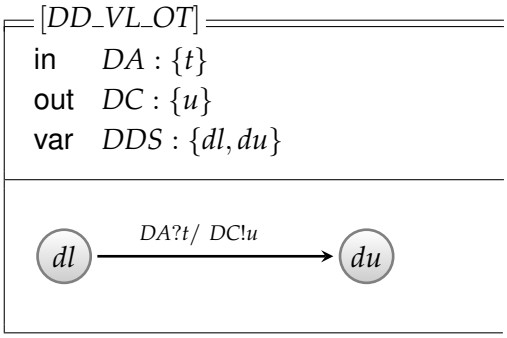
This corresponds to Moore-Automata [Moore, 1956]. The outputs must only depend on inputs that the service receives earlier and the state the service stats in. Therefore, the function only reads inputs in the first time interval. The contents of the output channels in this first time interval are arbitrary and not controlled by the service. The "$*$" in the stream in the figure indicates arbitrary values. The actual values have no effect on the transition. Actually a "$*$" represents a set of semantic tuples, each element having one of all possible values on these channels.

The situation is similar, in the second time interval. Only the outputs are relevant for the service i.e. the outputs are controlled. The I/O relation of the service defines them. The inputs are arbitrary (except for those inputs that are part of a feedback loop) and appear with any values in the semantic tuples. In fact the service does not receive the inputs any more. The states of the semantic tuples represent the history of earlier inputs (i.e., they are equivalence classes for these inputs).

EXAMPLE 4.3 (SEMANTIC MAPPING OF ATOMIC SERVICES)
*In Figure 4.2(a) the driver door switches to state "unlocked" (*du*) if the signal* t *arrives on channel* DA *and the control state* DDS *is "locked" (*dl*). The transition causes a message* u *as an instruction to physically unlock the door.*

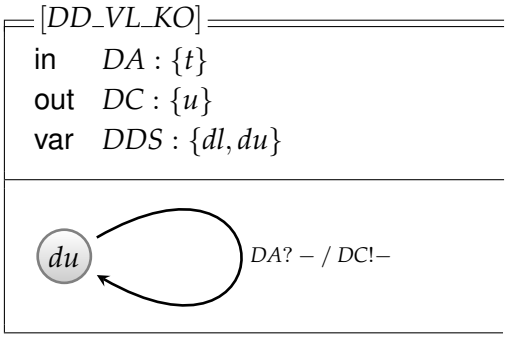
Figure 4.2(a) is a simple service with a single transition such that the pre-condition and the post-condition do not match. Hence, a repeated execution of the transition is not possible. As a consequence, only a single semantic tuple is in its set of behaviors as depicted in a simplified way below the figure.

*In Figure 4.2(b) the system is in control state "unlocked" (*du*). As long as no signal arrives the state remains unchanged and no outputs appear.*



$$\{(du, \begin{matrix}\langle\!\langle \langle t\rangle\ \langle\bullet\rangle \rangle\!\rangle, \\ \langle\!\langle \langle\bullet\rangle\ \langle u\rangle \rangle\!\rangle\end{matrix}, dl)\}$$

(a) Unlocking the driver door on receiving a touch event (cf. $F_{7,d}$ of the case study)



$$\{(du, \begin{matrix}\langle\!\langle \langle -\rangle\ \langle\bullet\rangle \rangle\!\rangle, \\ \langle\!\langle \langle\bullet\rangle\ \langle -\rangle \rangle\!\rangle\end{matrix}, du),$$
$$(du, \begin{matrix}\langle\!\langle \langle -\rangle\ \langle -\rangle\ \langle\bullet\rangle \rangle\!\rangle, \\ \langle\!\langle \langle\bullet\rangle\ \langle -\rangle\ \langle -\rangle \rangle\!\rangle\end{matrix}, du), \dots\}$$

(b) Keep driver door unlocked if no input is received (cf. $F_{7,d}$ of the case study)

Figure 4.2.: A clipping from the case study illustrating single transitions and their semantic mapping

In contrast to 4.2(a), Figure 4.2(b) describes a transition where the post-condition satisfies the pre-condition. As a consequence, the set of behaviors contains semantic tuples with a growing length of I/O-histories. This is a consequence of the closure operator in the semantic mapping.

Below the specifications the figures present a simplified representation of the semantic tuples: Since the states are only built on control states, we omit the variable names and only give their values. ♣

The closure operator creates any possible sequence of interactions out of the defined patterns by pasting together behaviors that have matching final and initial states. The interaction patterns of the input and output channels are pasted overlapping to handle restricted causality and to produce appropriate behaviors. The first service's final outputs affect the same time interval where the second service starts reading the inputs. Remember the definition of function $\frown$ () in Section 2.2 as it is used in the definition of the closure $\mathcal{CL}()$. The operation pastes two streams such that the last time interval of the first stream and the first time interval of the second stream overlap. With the inputs being arbitrary for the first stream and the outputs being arbitrary for the second stream this usually works well.

Note that this operation in general is partial. Even if the states are compatible, it still is possible that feedback channels contain values that are not in the second service's domain. The closure operation only combines tuples with compatible valuations at feedback channels. Figure 4.3 illustrates the situation.

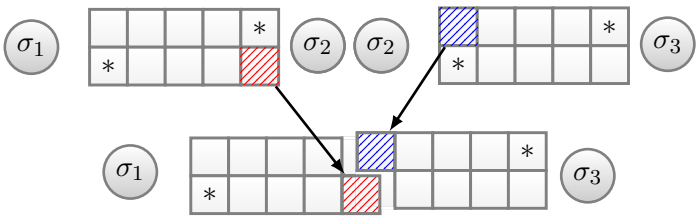


Figure 4.3.: Overlapping pasting of behaviors. The outputs at one time interval must coincide with the inputs at the next time interval in the case of feedback.

According to the definition of atomic services and their semantics, functions can be partial. According to [Schätz and Salzmann, 2003] there are two possible formal interpretations of under-specification:

1. *no behavior* is possible, if there is no explicit specification (closed world assumption)[Reiter, 1987]

2. *any behavior* is possible, if there is no explicit specification (open world assumption)

The semantics of MARLIN assigns the empty set of outputs to under-specified inputs. This is a closed world assumption and aligns with the discussion in [Schätz and Salzmann, 2003].

4.4. Binary compositions

Use-cases are usually unrelated at the beginning of a development and need to be composed. This applies most to use-cases describing the core system (cf. Section 3.1) of a context-adaptive system because the core system contains the observable behaviors. Subsequently, we describe alternative and parallel composition to compose behaviors without conditions. Both operators base on ideas of Schätz [Schätz, 2007] and [Schätz,

2009]. However, their definitions in MARLIN differ from the mentioned approaches in some important details. According to the big step semantics, they can define behaviors of arbitrary length – which we later need in the definition of mode transition systems – and preserve infix closure.

4.4.1. Alternative composition

Alternative composition combines two services such that the new service provides either of both behaviors. Without restriction, we regard the alternative composition of services A and B. The composed service's interface is the join of the interface of the original services and provides the behaviors that are either valid for A or for B.

Definition of alternative composition Alternative composition extends behaviors a) by adding system reactions for yet unconsidered inputs (reducing partiality) and b) by adding additional alternatives (raising the level of non-determinism). The idea to use alternative composition to extend behaviors to longer sequences bases on ideas originally presented by Pnueli and Manna [Manna and Pnueli, 1995] and explained in more detail in [Clarke et al., 1999].

DEFINITION 23 (ALTERNATIVE COMPOSITION):
Let $A \in \mathbb{F}(I_A \rhd O_A, V_A)$ and $B \in \mathbb{F}(I_B \rhd O_B, V_B)$ be two services with possible intersecting sets of inputs, outputs, and variables. To prevent an unintended interference, we require for both services that their respective set of mode variables is empty:

$$VM_A = VM_B = \varnothing$$

The interface and variables of a service $S \stackrel{def}{=} A \oplus B$ are:

- $I_S = I_A \cup I_B$
- $O_S = O_A \cup O_B$
- $VC_S = VC_A \cup VC_B$

The behavior of the compound service is:

$$\begin{aligned}[\![A \oplus B]\!] \stackrel{def}{=} \mathcal{CL}.\{&(\sigma, i, o, \acute{\sigma}) \mid \\ &(\sigma, i, o, \acute{\sigma})_{\lfloor A} \in [\![A]\!] \quad \vee \\ &(\sigma, i, o, \acute{\sigma})_{\lfloor B} \in [\![B]\!] \}\end{aligned}$$

Note that the alternative composition maintains the three properties of restricted causality, infix closure and behavioral closure. For the proofs see Sections A.1 and A.2. □

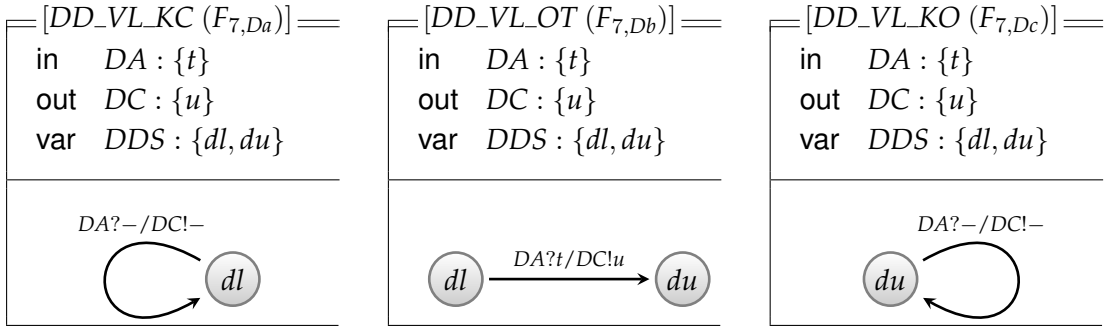
The restriction operator for variables and channels in the definition allows arbitrary values for those resources that are not explicitly controlled by a service. This introduces non-determinism for those resources.

For mode transition systems we will not allow uncontrolled non-determinism. This counteracts structural features that we exploit for restructuring and analyzing the specifications. If mode transition system variables are exclusive to B and A is active, the variables can change arbitrarily. As a result, the mode transition system is in an arbitrary state which causes, e.g., uncontrolled mode hopping.

By requiring an empty set of the both services' mode-variable sets ($VM_A = VM_B = \varnothing$) we effectively disallow the alternative composition of services involving mode transition systems. Intended non-determinism for mode changes can be defined explicitly in the set of mode transitions as described later.

The constructive nature of alternative composition Alternative composition is a constructive way to build up functionality. In a state, the compound service is able to react on inputs according the specification of either service possibly leading to alternative successor states. If both services offer options for a proper reaction, one option is chosen non-deterministically. If none of both services is capable of reacting to the input, the behavior is (still) undefined in the compound service. In the following example we show how alternative composition adds behavior to a specification.

EXAMPLE 4.4 (ALTERNATIVE COMPOSITION OF THREE SERVICES)
We build a more complex behavior `DD_VL_OPEN` *from atomic services* `DD_VL_KC`, `DD_VL_OT` *and* `DD_VL_KO` *by alternative composition. The three services appear in Figure 4.4 (Additional names in brackets are according to Appendix B).*



(a) Service `DD_VL_KC`: The driver door keeps closed if no event is received

(b) Service `DD_VL_OT`: The driver door is opened if a touch event is received

(c) Service `DD_VL_KO`: The driver door stays open if no event is received

Figure 4.4.: Three atomic services contributing to the driver door function `DD_VL_OPEN`

The alternative composition of the three services returns the service DD_VL_OPEN *in Figure 4.5.*

[DD_VL_OPEN]

$$DD\_VL\_OPEN \stackrel{def}{=} DD\_VL\_KK \oplus DD\_VL\_OT \oplus DD\_VL\_KO$$

Visualized as an automaton the service looks as follows:

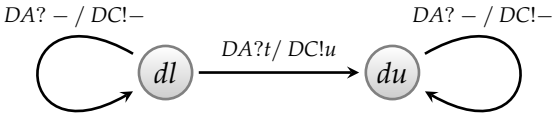


Figure 4.5.: The composed service DD_VL_OPEN

Note that the automaton representation is only an illustration. The actual composition does not structurally compose the services but only their behaviors. ♣

The transitive closure of alternative composition The definition of alternative composition of behaviors uses the closure operation instead of just joining the sets of behaviors of either operand. One reason is the extension of possible interaction patterns due to new options for interactions. Another reason is the interleaving of behaviors. Alternative compositions allows choosing between A and B in every execution step.

We discuss this in more detail. Assume the sets of behaviors are joined without the use of the closure. After the activation of a behavior, the system chooses (possibly non-deterministically) a behavior of one of the sub-services. Without restriction, assume the choice has been service A. Service A reacts to inputs for a while (say until t) after which it receives an input (m) that A cannot react to. Let σ_t be the state of A at time t. Even if B provides proper reactions for a state σ_t and the considered input m at t a switch is impossible because the joined semantic tuples are specific for A or B only.

By applying the closure we address this issue. As demonstrated in Theorem 12 all services are infix closed. Hence, the history of A until time t with σ_t as a final state is in the set of valid behaviors of A. Furthermore, the history starting in σ_t and accepting the next input is a valid behavior of B. The closure creates longer behaviors out of these behaviors. This effectively allows alternating between the two services. Furthermore, the compound service again is infix closed (cf. Theorem 12). The following example illustrates the difference.

EXAMPLE 4.5 (JOIN WITHOUT CLOSURE)
Figure 4.6 presents two behaviors acting on the same variable $v : \{a, b, c\}$ *and using the same inputs and outputs* $i : \{1, 2, 3\}$ *and* $o : \{\alpha, \beta, \gamma, \delta\}$.

Note that the presentation as automaton is just for illustrative reasons. Actually we regard the sets:

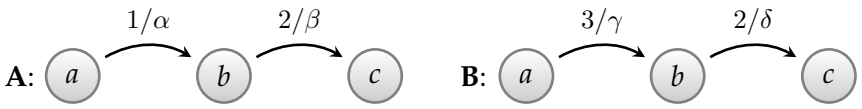


Figure 4.6.: The behaviors of services A and B illustrated in automata notation

A: $\{(a, \langle\!\langle\langle 3\rangle\langle\bullet\rangle\rangle\!\rangle, \langle\!\langle\langle\bullet\rangle\langle\gamma\rangle\rangle\!\rangle, b), (b, \langle\!\langle\langle 2\rangle\langle\bullet\rangle\rangle\!\rangle, \langle\!\langle\langle\bullet\rangle\langle\delta\rangle\rangle\!\rangle, c), (a, \langle\!\langle\langle 3\rangle\langle 2\rangle\langle\bullet\rangle\rangle\!\rangle, \langle\!\langle\langle\bullet\rangle\langle\gamma\rangle\langle\delta\rangle\rangle\!\rangle, b)\}$

B: $\{(a, \langle\!\langle\langle 1\rangle\langle\bullet\rangle\rangle\!\rangle, \langle\!\langle\langle\bullet\rangle\langle\alpha\rangle\rangle\!\rangle, b), (b, \langle\!\langle\langle 2\rangle\langle\bullet\rangle\rangle\!\rangle, \langle\!\langle\langle\bullet\rangle\langle\beta\rangle\rangle\!\rangle, c), (a, \langle\!\langle\langle 1\rangle\langle 2\rangle\langle\bullet\rangle\rangle\!\rangle, \langle\!\langle\langle\bullet\rangle\langle\alpha\rangle\langle\beta\rangle\rangle\!\rangle, b)\}$

We now compare the set of possible behaviors according to conventional language join and according to the actual definition of alternative composition in MARLIN.

Conventional language join

$$\{(a, \langle\!\langle\langle 3\rangle\langle\bullet\rangle\rangle\!\rangle, \langle\!\langle\langle\bullet\rangle\langle\gamma\rangle\rangle\!\rangle, b), (b, \langle\!\langle\langle 2\rangle\langle\bullet\rangle\rangle\!\rangle, \langle\!\langle\langle\bullet\rangle\langle\delta\rangle\rangle\!\rangle, c), (a, \langle\!\langle\langle 3\rangle\langle 2\rangle\langle\bullet\rangle\rangle\!\rangle, \langle\!\langle\langle\bullet\rangle\langle\gamma\rangle\langle\delta\rangle\rangle\!\rangle, b),$$
$$(a, \langle\!\langle\langle 1\rangle\langle\bullet\rangle\rangle\!\rangle, \langle\!\langle\langle\bullet\rangle\langle\alpha\rangle\rangle\!\rangle, b), (b, \langle\!\langle\langle 2\rangle\langle\bullet\rangle\rangle\!\rangle, \langle\!\langle\langle\bullet\rangle\langle\beta\rangle\rangle\!\rangle, c), (a, \langle\!\langle\langle 1\rangle\langle 2\rangle\langle\bullet\rangle\rangle\!\rangle, \langle\!\langle\langle\bullet\rangle\langle\alpha\rangle\langle\beta\rangle\rangle\!\rangle, b)\}$$

Alternative composition

$$\{(a, \langle\!\langle\langle 3\rangle\langle\bullet\rangle\rangle\!\rangle, \langle\!\langle\langle\bullet\rangle\langle\gamma\rangle\rangle\!\rangle, b), (b, \langle\!\langle\langle 2\rangle\langle\bullet\rangle\rangle\!\rangle, \langle\!\langle\langle\bullet\rangle\langle\delta\rangle\rangle\!\rangle, c), (a, \langle\!\langle\langle 3\rangle\langle 2\rangle\langle\bullet\rangle\rangle\!\rangle, \langle\!\langle\langle\bullet\rangle\langle\gamma\rangle\langle\delta\rangle\rangle\!\rangle, b),$$
$$(a, \langle\!\langle\langle 1\rangle\langle\bullet\rangle\rangle\!\rangle, \langle\!\langle\langle\bullet\rangle\langle\alpha\rangle\rangle\!\rangle, b), (b, \langle\!\langle\langle 2\rangle\langle\bullet\rangle\rangle\!\rangle, \langle\!\langle\langle\bullet\rangle\langle\beta\rangle\rangle\!\rangle, c), (a, \langle\!\langle\langle 1\rangle\langle 2\rangle\langle\bullet\rangle\rangle\!\rangle, \langle\!\langle\langle\bullet\rangle\langle\alpha\rangle\langle\beta\rangle\rangle\!\rangle, b),$$
$$(a, \langle\!\langle\langle 3\rangle\langle 2\rangle\langle\bullet\rangle\rangle\!\rangle, \langle\!\langle\langle\bullet\rangle\langle\gamma\rangle\langle\beta\rangle\rangle\!\rangle, c), (a, \langle\!\langle\langle 1\rangle\langle 2\rangle\langle\bullet\rangle\rangle\!\rangle, \langle\!\langle\langle\bullet\rangle\langle\alpha\rangle\langle\delta\rangle\rangle\!\rangle, c)\}$$

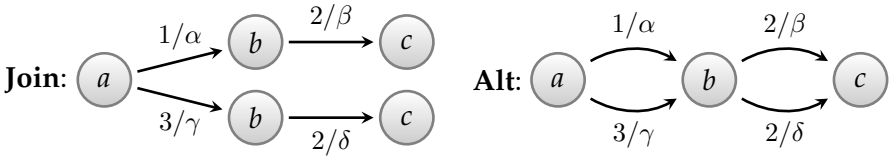


Figure 4.7.: Comparison of conventional join and alternative composition

The elements $(a, \langle\!\langle\langle 3\rangle\langle 2\rangle\langle\bullet\rangle\rangle\!\rangle, \langle\!\langle\langle\bullet\rangle\langle\gamma\rangle\langle\beta\rangle\rangle\!\rangle, c)$ *and* $(a, \langle\!\langle\langle 1\rangle\langle 2\rangle\langle\bullet\rangle\rangle\!\rangle, \langle\!\langle\langle\bullet\rangle\langle\alpha\rangle\langle\delta\rangle\rangle\!\rangle, c)$ *are exclusive elements of the composed behavior according to alternative composition in* MARLIN *because there is a shared state b that allows a switch between the two services. Figure 4.7 illustrates the difference in the automata notation.* ♣

4.4.2. Parallel composition

Parallel composition combines two services simultaneously: the compound service provides both behaviors at the same time. Without restriction, we regard the parallel composition of services A and B. The interfaces, including the variables, of both services are joined. Services represent views on a system which may concern overlapping aspects.

Definition of parallel composition In the definition we use an auxiliary function that prepares the set of behaviors similar to the auxiliary function $\mathcal{CL}$.

$$\begin{aligned}&\mathcal{H} \in \mathcal{P}(\mathbb{T}) \to \mathcal{P}(\mathbb{T}) :\\ &\quad \mathcal{H}.X \mapsto Y \text{ such that}\\ &\qquad Y = \nu Y.X \cap \{(\sigma, i_1 \frown i_2, o_1 \frown o_2, \acute{\sigma}) \mid \exists \sigma_l \in \Sigma, t \in \mathbb{N} : t \geq 2 \wedge\\ &\qquad\qquad (\sigma,\ i_1,\ o_1,\ \sigma_l) \in Y \wedge\\ &\qquad\qquad (\sigma_l,\ i_2,\ o_2,\ \acute{\sigma}) \in Y \}\end{aligned}$$

The function takes a set of behaviors as an input and returns a subset with a special property: all behaviors can be cut into arbitrary pieces such that they are valid behaviors as well. We may think of $\mathcal{H}$ as a kind of complement for $\mathcal{CL}$ although it is not its reverse function. Essentially, the function $\mathcal{H}$ selects a subset (the largest subset) of a set of behaviors that is infix closed.

DEFINITION 24 (PARALLEL COMPOSITION):
Let $A \in \mathbb{F}(I_A \rhd O_A, V_A)$ and $B \in \mathbb{F}(I_B \rhd O_B, V_B)$ be two services that are combined into a service $S = A \otimes B$. The interface of the compound service S is defined as:

- $I = (I_A \cup I_B)$
- $O = O_A \cup O_B$
- $VC = VC_A \cup VC_B$
- $VM = VM_A \cup VM_B$

For a parallel composition there are no restrictions regarding the sets of channels and variables.

In the definition we use the auxiliary channel set $C = I_A \cup I_B \cup O_A \cup O_B$. The behaviors of the service $A \otimes B$ are defined by:

$$\begin{aligned}[\![A \otimes B]\!] \stackrel{def}{=}\ \mathcal{H}.\{\ &(\sigma, i, o, \acute{\sigma}) \mid \exists z \in \mathbb{H}(C) :\\ &z_{\lfloor I} = i \wedge z_{\lfloor O} = o \qquad\qquad \wedge\\ &(\sigma_{\lfloor V_A}, z_{\lfloor I_A}, z_{\lfloor O_A}, \acute{\sigma}_{\lfloor V_A}) \in [\![A]\!] \quad \wedge\\ &(\sigma_{\lfloor V_B}, z_{\lfloor I_B}, z_{\lfloor O_B}, \acute{\sigma}_{\lfloor V_B}) \in [\![B]\!] \ \}\end{aligned}$$

Note that the alternative composition maintains the three properties of restricted causality, infix closure and behavioral closure. For the proofs see Sections A.1 and A.2. □

To capture communication the definition relates outputs of one service with inputs of the other. The auxiliary channel history z over all channels $C = I_A \cup I_B \cup O_A \cup O_B$ fixes the contents of the communication channels from A to B and vice versa (if present). The rest of z complies with the actual inputs and outputs of the original service: $(z_{\lfloor I} = i) \wedge (z_{\lfloor O} = o)$.

The restrictive nature of parallel composition The nature of parallel composition restricts existing functionality and integrates different functions while reducing nondeterminism and possibly introducing partiality. We explain the nature of parallel composition subsequently and give examples. Without restrictions, for the subsequent considerations let:

$R_A = VM_A \cup VC_A \cup I_A \cup O_A$ // resources controlled by A
$R_B = VM_B \cup VC_B \cup I_B \cup O_B$ // resources controlled by B
$co = R_A \cap R_B$ // resources controlled by both services
$ex_A = R_A \setminus R_B$ // resources exclusive to A
$ex_B = R_B \setminus R_A$ // resources exclusive to B

Parallel composition uses the join of its operand services' interfaces as the interface for the compound service. In practice, parallel composition may include exclusive as well as shared resources. This allows modeling settings where two services use shared resources as a result of these services representing different (but overlapping) views.

Since A and B control the values of the resources in co directly, the compound services can only contain behaviors that are valid for both original services at the same time. Any semantic tuples that contain outputs or states that are unacceptable for one of the services are excluded from the composed service's set of accepted behaviors.

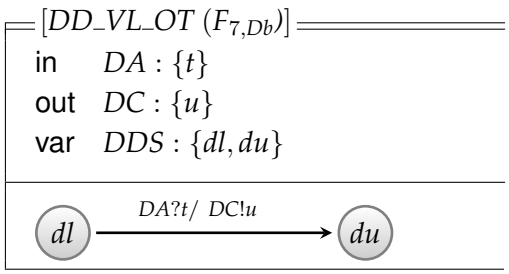
The intersection of behaviors may even be empty for some inputs and states. We say that the two services contradict with respect to some inputs. As a result, the composition is more partial with respect to these inputs, i.e., the domain of the compound service is smaller than the domains of the original services (with respect to proper projections of the compound service's domain).

In accordance to [Zave and Jackson, 2000] we call the control of shared resources *feature interactions*. Feature interactions that introduce additional partiality are "unwanted feature interactions". In Chapter 5 we discuss the discovery and treatment of unwanted feature interactions.

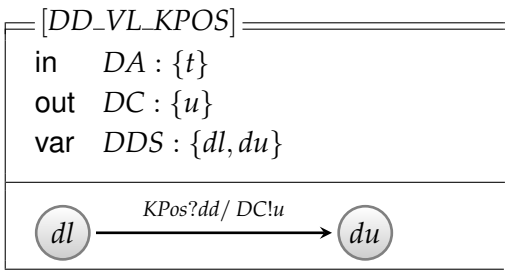
EXAMPLE 4.6 (RESTRICTION OF FUNCTIONALITY)
Figure 4.8 shows another parallel composition of two services from the case study. Figure 4.8(a) is the basic service that allows unlocking the driver door if receiving an appropriate signal. In addition, Figure 4.8(b) is a service that allows the unlocking only if a key is in range[1]*. The composed service is in Figure 4.9 and describes a service that unlocks the driver door only if a key is in range and the door activator is triggered.*

The composed service does not accept inputs that are accepted by only one of the original services. A touch event without a valid key is an example. Service 4.8(a) accepts this input and defines a proper reaction, since the projection to channel `DA` *is in the service domain. The*

[1] This service is actually not part of the case study as presented here. The availability of the key is modeled in a different way. The service `DD_VL_KPOS` is only for demonstrating the parallel composition in an illustrative manner



(a) Unlocking the driver door if an according request is received



(b) Unlocking is only permitted if a key is available at the driver door

Figure 4.8.: Tho atomic services describing two aspects of the door opening that are valid at the same time

composed service, in contrast, is unable to react to this input due to restrictions of Service 4.8(b). This is an unwanted feature interaction.

[DD_VL_OT_KEY]

$$DD\_VL\_OT\_KEY \stackrel{def}{=} DD\_VL\_OT \otimes DD\_VL\_KPOS$$

Visualized as an automaton the service looks as follows:

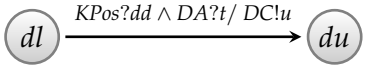


Figure 4.9.: Example of parallel composition of two services in the case study restricting the unlocking of the driver door only if a key is available

Note that the automaton representation is only an illustration. The actual composition does not structurally compose the services but only their behaviors. ♣

This restrictive nature applies to different resources as well. Again, the restriction operation for channels and variables in the definition allows arbitrary valuations of ex_B from the viewpoint of A. This corresponds to non-determinism of A with respect to ex_B. The same holds for B and ex_A.

For all resources the parallel composition has to satisfy the requirements of both operand services at the same time. The resulting behavior contains the *intersection* of possible valuations. For the resources ex_A and ex_B services B and A respectively make no restrictions. As a result, the intersection contains exactly appropriate valuations for A in the case of ex_A and for B in the case of ex_B.

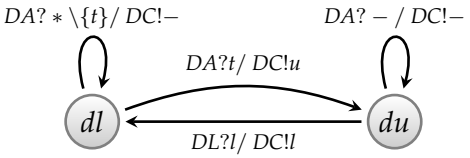
In Figure 4.10 service A restricts the valuations of resources in ex_A from being arbitrary to be like A defines them. The same holds for B and ex_B. An arbitrary behavior

A	B	$A \otimes B$	
$*$	b_1	b_1	ex_b
$*$	$\dots$	$\dots$	
a_1	b_j	$a_1 \cap b_j$	co
$\dots$	$\dots$	$\dots$	
a_i	b_m	$a_i \cap b_m$	
$\dots$	$*$	$\dots$	ex_a
a_n	$*$	a_n	

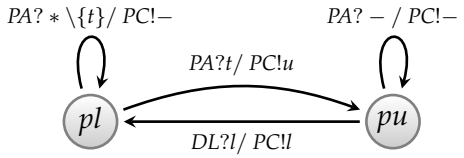
Figure 4.10.: Restriction of resources by A and B

is neural in the intersection. Thus, formally parallel composition always reduces the set of possible observations, even if we use it methodologically to extend an existing function by another one.

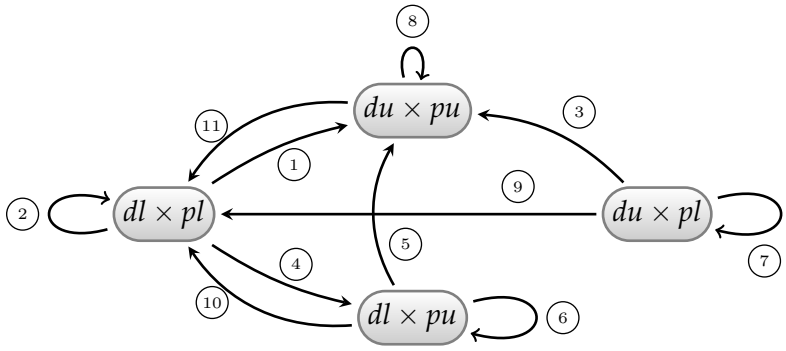
EXAMPLE 4.7 (INTEGRATION OF FUNCTIONALITY)
To show the integrative capabilities we compose the driver door without its side effects on the other doors, (Figure 4.11(a)) and the passenger door (Figure 4.11(a)). For reasons of the illustration we use the representation in an automaton style without the surrounding frame of the specification.



(a) (Un)locking the driver door if an according request is received



(b) (Un)locking the passenger door if an according request is received



(c) The parallel composition of both services

Figure 4.11.: Example of parallel composition of two services integrating functions for unlocking the driver door and unlocking the passenger door

We present the composition of the two functions in Figure 4.11(c) with the following mapping of transition labels:

① : $DA?t, PA?-/\ DC!u, PC!u$ ② : $DA?-, PA?-/\ DC!-, PC!-$
③ : $DA?-, PA?t/\ DC!-, PC!u$ ④ : $DA?-, PA?t/\ DC!-, PC!u$
⑤ : $DA?t, PA?-/\ DC!u, PC!-$ ⑥ : $DA?-, PA?-/\ DC!-, PC!-$
⑦ : $DA?-, PA?-/\ DC!-, PC!-$ ⑧ : $DA?-, PA?-/\ DC!-, PC!-$
⑨, ⑩, ⑪ : $DL?l/\ DC!l, PC!l$

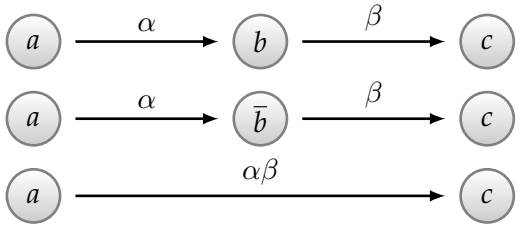
The compound service offers both functions: opening the driver door and opening the passenger door. We neglect the driver door's side effect to unlock any other doors here. This would cause a conflict and conflict identification and resolution are addressed later (refer to Section 5.2.1.2 for further details on that topic). ♣

The infix closure of parallel composition Intersecting behaviors has a side effect on the semantic level that we need to take care of. This side effect is a matter of the semantics and no matter of the system under construction.

The function $\mathcal{H}$ ensures that it is possible to cut all behaviors in the set of behaviors of the compound service into shorter behaviors that are valid for the compound service as well. We call this property an infix closure. The property is mandatory for defining the conditional composition in Section 4.5.

As an issue, the two operand services could take a conflicting intermediate state. Without $\mathcal{H}$ such a behavior would be in the intersection of the behaviors but the related parts were not. As a result the compound service is unrealizable. To provide the behavior the compound service needs to take a conflicting intermediate state. We illustrate this with an abstract example.

EXAMPLE 4.8 (PARALLEL COMPOSITION WITHOUT $\mathcal{H}$)
To show the effect of $\mathcal{H}$ we give an example of the parallel composition of two abstract services. We use b and $\overline{b}$ to indicate that both services take conflicting states after an abstract I/O event α. Therefore, the behaviors using these states are not part of the composed service.



Without applying the function $\mathcal{H}$ the big transition from state a to state c with the I/O $\alpha\beta$ remains. In fact this cannot be implemented since the intermediate state leading to the remaining behavior does not exist. The function $\mathcal{H}$ removes this behavior as well because it is no longer the result of a closure operation of existing (shorter) behaviors. ♣

4.5. Conditional composition

In Chapter 3 we define context-adaptive systems as systems with I) a set of functions each satisfying different requirements for certain contexts and directly interacting with users (called the core system) and II) a logic that selects the appropriate functionality in accordance to the observed context (called the adaptation sub-system). We continue with conditional composition that specifies the adaptation sub-system of context-adaptive systems [2].

Conditional composition composes services such that their effectiveness depends on additional conditions. In MARLIN mode transition systems (mts) realize the conditional composition. In this section we define the syntax and semantics of mode transition systems. We start by recalling the syntax of mode transition systems and the syntax of the contained elements and continue with considerations about the interface. Subsequently we discuss aspects of interrupting and resuming behavior. This information is the prerequisite to defining the semantics of mode transition systems. We finish by relating the semantics of mode transition systems to the aspects of interrupting and resuming behaviors.

4.5.1. Modes and their aspects

Mode transition systems are labeled transition systems with nodes that we call modes and guarded transitions between these modes. Modes gather services and capture the behavior of the system in certain situations. Mode transitions determine the switch between services and realize the conditions of activation. The main issues of mode transition systems comprise aspects and terms of suspending and resuming behaviors.

4.5.1.1. Constituents of mode transition systems

Mode transition systems are basically conventional labeled transition systems. This fits to well-analyzed and understood concepts of system modeling. Especially, associating behaviors with nodes is similar to Moore-automata [Moore, 1956]. The difference is that states in mode transition systems may contain any complex I/O-behavior not only a single defined output. Modes are abstract, hierarchical states that combine a number of lower-level states and transitions into a black box. As a graphical representation, similar to Moore-automata we use a split node showing a mode's name and the associated service.

Formally, a mode transition system is a tuple $(\mathbb{M}, \delta, \Phi)$ which defines the modes, the transitions between the modes and their associated behavior.

$\mathbb{M}$ is a set of mode names.

[2] In addition to specifying the adaptation sub-system, conditional composition is useful for resolving unwanted feature interactions. We discuss feature interactions and their relation to context adaptation in Chapter 5.

δ is the set of transitions between nodes. A transition relates two modes and a transition label. The transition label is a condition on streams and evaluates the activation of the transition.

$$\delta : \mathbb{M} \times ((\mathbb{H}(I \cup O) \to \mathbb{B}) \times \{\epsilon, \rho\}) \to \mathbb{M}$$

We allow conditions to account for outputs as well. This enables the writing of concise labels. Formally, accounting only for inputs is similar expressive: any output is the result of a function of inputs. However, a restriction to inputs may involve a recreation of any complex algorithmic deduction of outputs.

We say that a mode is activated, if an enabled transition leads to this mode. The set $\{\epsilon, \rho\}$ is not involved in determining a successor directly but determines the evaluation of successive transition conditions. Their meanings are *reset* and *pursue*.

We define the set of successors of a mode:

$$succ : \mathbb{M} \to \mathcal{P}(\mathbb{M});$$
$$succ(m) = \{m' \mid \exists\, (c, s) : (m, (c, s), m') \in \delta\}$$

Φ is a function $\Phi : \mathbb{M} \to ID_S$ that maps mode names to services. For a mapping $m \mapsto f$ we call f the mode-behavior of m.

A mode transition system again defines a service. We say that a service is in mode m if the service is a mode transition system and the respective mode m is active in this mode transition system.

Mode transition systems allow separating the descriptions of the mode-behaviors and the logic that determines their activation. This corresponds to the definitions of context-adaptive systems in Chapter 3.1 where we separated the system-core and the adaptation logic to support the understanding of the definition. In a real system specification context often only affects some aspects of the system. Therefore, it is possible to combine parallel composition and mode transition systems. The result is a distributed adaptation logic. Section 6.2.2 presents a transformation that disentangles the specification to reveal the clear separation of both.

EXAMPLE 4.9 (SIMPLE MODE TRANSITION SYSTEM)
The specification `Example_MTS` *shows a service that is a simple mode transition system with two modes m_a and m_b and transitions triggered by inputs on a channel i. The mode-behaviors are A and B without further detailing them. Figure 4.12 shows the mode transition system in automata style.*

[*Example_MTS*]

$$\begin{aligned} \mathit{Example\_MTS} \stackrel{def}{=}\ & \{m_a, m_b\} \\ & \{(m_a, (i = [\![\langle\bullet\rangle^* \langle T\rangle]\!], \rho), m_b), (m_b, (i = [\![\sharp(\overline{\langle F\rangle})]\!], \rho), m_a)\} \\ & \{(m_a, A), (m_b, B)\} \end{aligned}$$

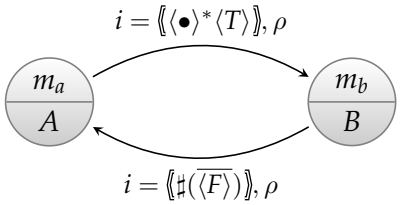


Figure 4.12.: Simple mode transition system with two modes. Representation in automata style without surrounding specification frame

The transition from m_a to m_b has a label $(i = \langle\!\langle \langle\bullet\rangle^* \langle T\rangle \rangle\!\rangle, \rho)$. This represents a sequence of any inputs ending with a message T. Hence, if the service receives a message T in mode m_a it switches to mode m_b. A condition $(i = \langle\!\langle \sharp(\overline{\langle F\rangle}) \rangle\!\rangle, \rho)$ enables the transition from m_b to m_a in a similar way. ♣

Syntax of mode transition systems We repeat the syntax for specifying mode transition systems in MARLIN:

<S> ::= (<M>,<δ>,<Φ>)

Let $ID_M \subseteq NAMES$ be a set of mode names such that $ID_S \cap ID_M = \varnothing$. Subsequently we use $m_1, m_2 \ldots$ or $m, m', m'', \ldots$ or $m_a, m_b, \ldots$ as representatives of these names. The non-terminals in the above syntax definition are defined as:

<M> ::= {[<$\mathrm{ID_M}$>;]$^+$} //nonempty set of mode names
<δ> ::= {[(<$\mathrm{ID_M}$>,<L>,<$\mathrm{ID_M}$>);]*} //set of transitions, possibly empty
<Φ> ::= {[(<$\mathrm{ID_M}$>,<S>);]$^+$} // nonempty set of mappings from mode names to services

A mode transition system has at least one mode with an associated mode-behavior and an arbitrary number of mode transitions.

The transition label <L> has two parts:

<L> ::= (<C>,{$\epsilon|\rho$})

The syntactic domain <C> is an expression that matches the representation of channel histories as presented in Section 2.3. This introduces a restriction to the expressiveness of guards. They must be regular i.e. we can express them in a regular language. This ensures decidability and eases their handling. Especially the upcoming analysis in Section 6.2 relies on the decidability and the known measures of handling regular languages.

In addition, a transition's label has either a ϵ or a ρ flag. This flag controls the involvement of earlier inputs for evaluating transition conditions. We discuss their meaning below.

Interface of mode transition systems Let $(m, (c, l), m') \in \delta$ where c is a condition $\mathbb{H}(C_c) \to \mathbb{B}$. The channel set $C_c = I_c \cup O_c$ comprises a set of input channels – not necessarily related to the channels of any mode-behavior – and a set of output channels. This channel set carries the information that a transition guard c uses to decide if its related transition is enabled.

We denote the set of all channels C_M built from input channels I_C and output channels O_C used in any transition guard respectively as

$$C_M = \bigcup_{(m,(c,l),m') \in \delta} C_c \qquad I_C = \bigcup_{(m,(c,l),m') \in \delta} I_c \qquad O_C = \bigcup_{(m,(c,l),m') \in \delta} O_c$$

As a syntactic restriction we require that

$$\forall (m, (c, l), m') \in \delta : O_c \subseteq O_{\Phi(m)}.$$

Transition guards are only allowed to relate to output channels that are under control of the active mode's behavior. Hence, the transition guards contribute to the set of input channels but not to the set of output channels of the interface of a mode transition system.

A mode transition system $M = (\mathbb{M}, \delta, \Phi)$ in MARLIN implicitly maintains two dedicated variables. PC_M stores the current mode. Its type is $\mathbb{M} \in \mathcal{P}(ID_M)$. The variable h_M stores the history of channels associated with the evaluation of conditions. We define the type of h_M as $\mathbb{H}(C_M)$. We call these variables mode-pointer and history-variable respectively.

DEFINITION 25 (INTERFACE OF A MODE TRANSITION SYSTEM):
Let $M = (\mathbb{M}, \delta, \Phi)$ be a mode transition system definition as given above. The interface of M is:

- $I = (\bigcup_{m \in \mathbb{M}} I_{\Phi(m)}) \cup I_C$
- $O = \bigcup_{m \in \mathbb{M}} O_{\Phi(m)}$
- $VC = (\bigcup_{m \in \mathbb{M}} VC_{\Phi(m)})$
- $VM = (\bigcup_{m \in \mathbb{M}} VM_{\Phi(m)}) \uplus \{PC_M, h_M\}$

As a special restriction we require: $\forall\, m, n \in \mathbb{M} : VM_{\Phi(m)} = VM_{\Phi(n)}$ i.e. all mode-behaviors have the same set of mode-variables. We explain this restriction in Section 4.5.3. □

The input channels I of the mode transition system are the union of the input channels of all involved mode-behaviors and all input channels that occur in any transition-condition. The output channels O are the union of all output channels of all involved mode-behaviors. Similar, the core variables of all mode-behaviors are joined. Finally, the mode-variables of all mode-behaviors are joined and supplemented by the mode-variables of the current mode transition system.

4.5.1.2. Interruption and resumption of behavior

Replacing a current behavior by a new, for the current context more appropriate one, basically involves three steps:

1. Interrupting (or suspending) a current behavior
2. Selecting a new behavior based on available information about the environment and/or internal information.
3. (Re)starting the new behavior (possibly resuming it, if it has been interrupted in the past).

These steps lead to three issues that we discuss subsequently.

Time of interruption *Weak preemption* allows a behavior to reach a defined state i.e. it cannot be interrupted at any time but only at defined times. This ensures that nothing unintended happens as a consequence of the interruption. In contrast, *Strong preemption* (sometimes simply preemption) immediately interrupts any current behavior. Formally, strong preemption satisfies the property

$$(\sigma, i \gg j) \in dom(F) \Rightarrow (\sigma, i) \in dom(F).$$

For all valid behaviors a truncated valid behavior exists. For causal behaviors this corresponds to prefix closure.

Strong preemptive interruption is similar to scheduling in an operating systems which allows interruptions (almost) at any time. An interruption causes the storing of the current data- and control state. A later reactivation restores them, allowing continuing the behavior at any later time as if it never stopped.

In an operating system the interruption is self-triggered and the operating system has full control over the scheduling. A fair scheduler eventually reactivates an interrupted process. In mode transition systems the environment triggers interruptions. Therefore, it is uncertain that eventually some situation allows the reactivation of a behavior (the environment may be unfair). Furthermore, a mode transition system may be unfair itself. Once a mode loses the control, the structure of the mode transition system may not allow returning to that mode. We say that a mode transition system is fair iff

$$\forall\, m \in \mathbb{M} : m \in succ(m)^*$$

where $succ(m)^*$ is the transitive closure of $succ()$.

Scope of conditions A second aspect is the length of the considered I/O-histories for deciding about a mode change. Evaluating a *single time tick* covers only aspects that currently can be observed.

$$c : (ch_1 \to \mathbb{D} \times \ldots \times ch_n \to \mathbb{D}) \to \mathbb{B}.$$

The predicate maps the current inputs at a number of channels $ch_1, \ldots, ch_n$ to a truth value. This is the original idea of state transition diagrams that evaluate the current inputs to decide about the next state.

Broy [Broy, 1997] describes state transition diagrams that are able to take a finite (untimed) stream as an input.

$$c : (ch_1 \to \mathbb{D}^* \times \ldots \times ch_n \to \mathbb{D}^*) \to \mathbb{B}$$

The predicate maps *finite untimed* streams of messages within one time interval to a truth value. These inputs actually are a more complex type.

Evaluating a *time period* allows making statements about the past or about longer lasting time periods.

$$c : (ch_1 \to \mathbb{D}^{\underline{*}} \times \ldots \times ch_n \to \mathbb{D}^{\underline{*}}) \to \mathbb{B}$$

The predicate maps *finite timed* streams to a truth value.

Reactivation of behavior (Re)activating a previously interrupted behavior allows some scenarios for treating intermediate results. We discuss the most common ones.

Re-initialization resets intermediate results. A behavior always starts in the same (set of) state(s) regardless if a mode has been active before and intermediate results are available. All previous results and stored values get lost. A mode-behavior $F = \Phi(m)$ needs a set of initial states $\Lambda \in \mathcal{P}(\Sigma_F)$ such that

$$\forall(\sigma, x) \in dom(F) : \sigma \in \Lambda.$$

Informally spoken, only behaviors are valid that start in one of the defined initial states.

Persistence retains all intermediate results. Behaviors continue as if they never stopped. This usually is the way scheduling in operating systems works. Let $M = (\mathbb{M}, \delta, \Phi)$ be a mode transition system and Σ_M its state space. Let F be a service and $\sigma, \acute{\sigma} \in \Sigma_M$ states of the state space. For any mode $m \in \mathbb{M}$ where F is inactive i.e. F is no part of the mode-behavior:

$$(\sigma_{\lfloor V_{\Phi(m)}}, i, o, \acute{\sigma}_{\lfloor V_{\Phi(m)}}) \in [\![\Phi(m)]\!] \Rightarrow \sigma_{\lfloor V_F} = \acute{\sigma}_{\lfloor V_F}$$

Informally spoken, no service changes the variables of any inactive service.

Sharing makes intermediate results of the service available to sequel behaviors. Reactivating a service includes access to any intermediate results.

4.5.2. Semantics of mode transition systems

The definition of the semantics of mode transition systems is complex. To prepare the reader we discuss certain aspects of the definition before we present the complete definition.

The definition of mode transitions includes a recursion. The auxiliary relation F_m represents the behavior of a mode transition system that is in mode m. It takes care of the interactions with the mode-behaviors as well as of possible mode changes. If a mode transition system performs a mode change, there exists a time t such that I) the mode-behavior of mode m controls the behavior until t and II) the communication until t satisfies a transition condition and determines a successor mode. A mode change passes the control to the auxiliary function $F_{m'}$ according to this successor mode. The auxiliary function $F_{m'}$ handles the rest of the phase recursively continuing the behavior.

In nested mode transition systems the super-ordinate mode transition system can withdraw the control from the sub-ordinate mode transition system. Thus the sub-ordinate mode transition system has to handle finite behaviors. Therefore, we consider two cases for the definition of the auxiliary functions:

1. A mode transition can be performed – calling the auxiliary function of the successor mode and
2. Eventually, no transition condition can be performed in the reminder of the active phase and the now active function handles the rest of the phase.

We investigate both cases separately and strip the complex definition into a number of elementary expressions. We assign numbers to the expressions that allow their recognition in the complete definition.

In both cases we use an auxiliary function Θ that decides about enabling transitions at time t. Events that occurred in earlier phases can affect the decision. Therefore, Θ includes the complete history of messages observed at any relevant channel, even if these observations come from earlier phase of activity. The history variable h_M maintains this information.

$$\Theta : \Sigma \times (\mathbb{H}(I \cup O) \to \mathbb{B}) \times \vec{I} \times \vec{O} \to \mathbb{B};$$

$$\Theta(\sigma, c, i, o) \stackrel{def}{=} c\Big(\big[\sigma(h_M) \circ_t ((i \bowtie o)_{\lfloor C_M})\big]_{\lfloor C_c}\Big)$$

The operation $(i \bowtie o)_{\lfloor C_M}$ combines the channel histories i and o which correspond to the most recent observations and restricts them to the channels C_M. The resulting (restricted) channel history matches to the channel set of the history variable h_M. Therefore, it is possible to concatenate the channel histories stored in h_M and the recent information: $(\sigma(h_M) \circ_t ((x \bowtie y)_{\lfloor C_M}))$. Finally, each transition guard possibly refers to a subset of channels only. Therefore, Θ restricts the result again to the set of channels C_c that are relevant for evaluating a transition guard c and hands over the result to the transition guard c that is a Boolean function to decide if the channel history satisfies the condition.

No mode change is performed In the base case the auxiliary function F_m is responsible for providing a behavior for a phase that cannot be further decomposed into sub-phases. The following three conditions characterize a valid semantic tuple $(\sigma, i, o, \acute{\sigma})$ for the function F_m.

(1) None of the transition guards of the outgoing transitions of m are satisfied at any time:

$$\forall\, t \in [1, \#(i) - 1], \forall (m, (c, l), m') \in \delta : \neg(\Theta(\sigma, c, (i)_{\downarrow t}, (o)_{\downarrow t}))$$

Remember that the last element in i is arbitrary and serves as input to the next function (in this case the next mode of the super-ordinate mode transition system). Therefore, we exclude it from the investigation.

(2) The behavior must comply with the mode-behavior of the active mode. Therefore, the projection of any semantic tuple to the interface of the mode-behavior must be an element of the respective mode-behavior:

$$\exists\, \acute{\sigma} \in \Sigma : (\sigma, i, o, \acute{\sigma})_{\lfloor m} \in [\![\Phi_m]\!]$$

(3) The final state $\acute{\sigma}$ stores the mode transition system specific information. We require that the stored mode still is m if the mode transition system executes no transition. The history variable h_M is updated property with messages at channels C_M that arrived during the phase of execution.

$$\acute{\sigma}(PC_M) = m$$
$$\acute{\sigma}(h_M) = \sigma(h_M) \circ (i \bowtie o)_{\lfloor C_M}$$

One or more mode changes can be performed If some transition condition is valid at a time t, we decompose the considered phase into sub-phases. The auxiliary function F_m controls the decomposition. F_m determines the time t and the successor mode m' – and hence the next active auxiliary function $F_{m'}$ – based on the satisfied transition guard(s) at time t.

The behavior until time t complies with the current mode-behavior. The behavior after t is iteratively handled by the auxiliary function $F_{m'}$ until the rest of the phase matches the base case. The following four conditions characterize a valid semantic tuple $(\sigma, i, o, \acute{\sigma})$.

(4) F_m determines the time t where the I/O histories satisfy a transition guard. We require a minimal t, i.e., the mode transition system follows the first transition whose guard is satisfied. If multiple transitions are possible at the same minimal t the mode transition system chooses one non-deterministically. The input stream is only relevant until $\#(i) - 1$ because the last input belongs to the successor behavior.

$$\exists\, t \in [1, \#(i) - 1], (m, (c, l), m') \in \delta : \Theta(\sigma, c, (i)_{\downarrow t}, (o)_{\downarrow t})$$

(5) The function F_m controls the output until time $t+1$. This is due to the strictly causal nature of services: outputs (including any decision about mode changes) only become effective at time $t+1$.

For a proper transition of modes we require an intermediate state to hand over information to the next auxiliary function $F_{m'}$. The projection to the interface of the current mode's mode-behavior needs to comply with the current mode's mode-behavior.

$$\exists\, \sigma_l \in \Sigma : (\sigma, (i)_{\downarrow t+1}, (o)_{\downarrow t+1}, \sigma_l)_{\lfloor \Phi_m} \in ([\![\Phi_m]\!] \cup [\![ANY_1]\!])$$

To avoid yet another case we additionally employ a special abbreviation

$$[\![ANY_1]\!] \stackrel{def}{=} \{(\sigma, i, o, \acute{\sigma}) \mid \sigma = \acute{\sigma} \wedge i \in \mathbb{H}_1(I) \wedge o \in \mathbb{H}_1(O)\}$$

This special behavior covers situations where the reminder of a stream has length 1 i.e. the new mode becomes active but will not affect the systems behavior yet. We explain the operation of this abbreviation in more detail in Section 4.5.3 together with the general discussion of the semantic mapping.

(6) The previous expression already fixes the values of the intermediate state σ_l with respect to the mode-behavior's variables. The variable PC_M becomes m' according to the next active mode. If the transition label is

- ρ, the history-variable h_M is supplemented with the recent observations until time t allowing to pursue the history
- ϵ, the history-variable h_M is set to the empty stream, resetting the saved inputs and excluding them from any further condition evaluations.

$$\sigma_l(PC_M) = m'$$
$$(l = \rho) \Rightarrow (\sigma_l(h_M) = \sigma(h_M) \circ (((i \bowtie o)_{\lfloor C_M})_{\downarrow t}))$$
$$(l = \epsilon) \Rightarrow (\sigma_l(h_M) = \langle\rangle)$$

(7) A final expression handles the incremental pass of control. The behavior starting after time t is valid for the successor mode and the respective auxiliary function. In particular this allows further activations of transition guards at any later time according to $F_{m'}$.

$$(\sigma_l, (i)^{\uparrow t}, (o)^{\uparrow t}, \acute{\sigma}) \in [\![F_{m'}]\!]$$

Figure 4.13 illustrates the handover between two successor modes.

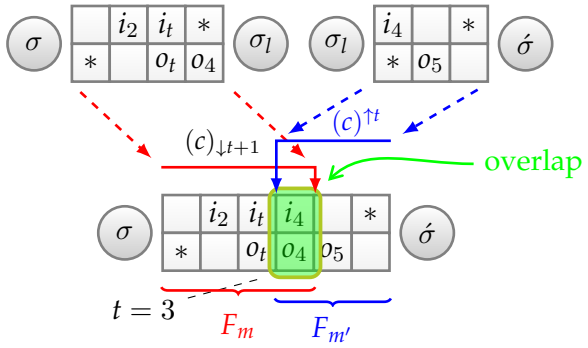


Figure 4.13.: Illustration of a semantic tuple built from function F_m and $F_{m'}$

The complete definition The above expressions describe the creation of the semantic tuples of a mode transition system if that is in a mode m, i.e., if $\sigma(PC_F) = m$. The complete definition reads:

DEFINITION 26 (BEHAVIOR OF A MODE TRANSITION SYSTEM IN MODE m):

$$
\begin{aligned}
[\![F_m]\!] = \{(\sigma, i, o, \acute{\sigma}) \mid\ & \forall t \in [1, \#(i) - 1], \forall (m, (c, l), m') \in \delta : \\
& \neg(\Theta_F(\sigma, c, (i)_{\downarrow t}, (o)_{\downarrow t})) && (1)\\
& \wedge\ (\sigma, i, o, \acute{\sigma})_{\lfloor \Phi_m} \in ([\![\Phi_m]\!] \cup [\![ANY_1]\!]) && (2)\\
& \wedge\ \acute{\sigma}(PC_F) = m\ \wedge \acute{\sigma}(h_F) = \sigma(h_F) \circ (i \bowtie o)_{\lfloor C_F}\} && (3)\\
\cup \{(\sigma, i, o, \acute{\sigma}) \mid\ & \exists t \in [1, \#(i) - 1], \sigma_l \in \Sigma, (m, (c, l), m') \in \delta : min(t) \\
& \wedge\ \Theta_F(\sigma, c, (i)_{\downarrow t}, (o)_{\downarrow t}) && (4)\\
& \wedge\ (\sigma, (i)_{\downarrow t+1}, (o)_{\downarrow t+1}, \sigma_l)_{\lfloor \Phi_m} \in ([\![\Phi_m]\!] \cup [\![ANY_1]\!]) && (5)\\
& \wedge\ \sigma_l(PC_F) = m' && (6)\\
& \wedge\ (l = \rho) \Rightarrow (\sigma_l(h_F) = \sigma(h_F) \circ (((i \bowtie o)_{\lfloor C_F})_{\downarrow t})) \\
& \wedge\ (l = \epsilon) \Rightarrow (\sigma_l(h_F) = \langle\!\langle\rangle\!\rangle) \\
& \wedge\ (\sigma_l, (i)^{\uparrow t}, (o)^{\uparrow t}, \acute{\sigma}) \in [\![F_{m'}]\!]\} && (7)
\end{aligned}
$$

In the definition $min(t)$ is an abbreviation for the formula

$$\nexists t' :\ t' < t\ \wedge\ \Theta_F(\sigma, c, (i)_{\downarrow t'}, (o)_{\downarrow t'}$$

It enforces a minimal t. □

The previous definition is just an auxiliary function that helps defining the mode transition systems' semantics. The function depends on the mode as a parameter. The definition of a mode transition system is independent of that parameter and receives the current mode via the state.

DEFINITION 27 (MODE TRANSITION SYSTEM):
A mode transition system examines the mode stored in the initial state and accordingly selects the auxiliary function F_m

$$[\![M]\!] = \{(\sigma, x, y, \acute{\sigma}) \mid \sigma(PC_M) = m \wedge (\sigma, x, y, \acute{\sigma}) \in [\![F_m]\!]\} \qquad \square$$

As an intuition, the service F changes the behavior according to the two cases. The current mode remains active until the inputs (or the input history) satisfy an outgoing transition. This fits to modes defining the behavior during phases of execution. Incoming transitions to a mode indicate the start of a phase and outgoing transitions indicate its end. We emphasize the relation between the phase character of modes and the fixed point semantics of the basic compositions. The mode-behavior reads the state at the beginning of the phase and behaves according to the inputs starting in this state while producing according outputs. Finally, the service exits the phase by properly setting a final state that serves as initial state for the successor mode.

4.5.3. Discussion of the semantics of mode transition systems

We discuss some important aspects of the definition and relate the discussion in Section 4.5.1.2 about aspects of mode transition systems to this definition. This especially is important in the context of nested mode transition systems. Therefore, we start with a small illustration of nested mode transition systems to support the understanding of their significance.

Nested mode transition systems The definition carefully handles the interruption of mode transition systems. Such interruptions only happen in nested mode transition systems. Example 4.10 presents a small nested mode transition system.

EXAMPLE 4.10 (NESTED MODE TRANSITION SYSTEMS)
Figure 4.14 illustrates a simple nested mode transition system. For reasons of simplicity we ignore the actual behaviors of the mode-behaviors `A`, `B` *and* `C` *and focus on the conditions of their activation. Furthermore, we assume that the conditions for the mode changes of the parent mode transition system* `M` *refer to a single channel* $\mathtt{C}_1$ *and the conditions of the child mode transition system* `N` *refer to a single channel* $\mathtt{C}_2$ *without going into details with the types of these channels.*

To describe the effects of the parent mode transition system interrupting the child mode transition system we assume conditions

$$c_1 = \{\!\{\langle\bullet\rangle^*\langle b\rangle\}\!\} \qquad c_2 = \{\!\{\langle\bullet\rangle^*\langle c\rangle\}\!\}$$
$$c_3 = \{\!\{(\langle\bullet\rangle^*\langle a\rangle)^3\}\!\} \qquad c_4 = \{\!\{\langle\bullet\rangle^*\langle d\rangle\}\!\}$$

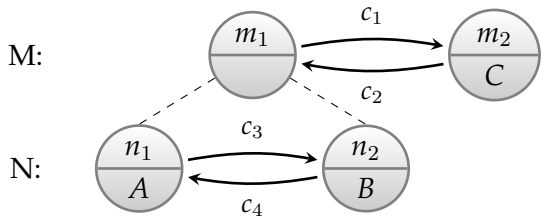


Figure 4.14.: A nested mode transition system. Representation in automata style without surrounding specification frame

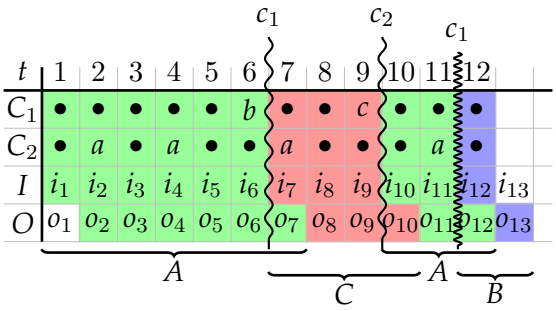


Figure 4.15.: Run of nested mode transition system where the parent mode transition system interrupts the child mode transition system before it receives a third message a

Figure 4.15 presents an example run. For an easier understanding, colors show which services receive inputs and which outputs they effectively control: green = A*, red =* C*, and blue =* B*.*

The mode transition systems are in modes m_1 *and* n_1 *at the beginning. Two messages* a *arrive on channel* C_2 *before a message* b *in channel* C_1 *causes the parent mode transition system to switch the mode to* m_2*.*

While in mode m_2 *any additional messages* a *have no effect on the child mode transition system (given that C is no mode transition system that affects the same mode-variables N). Actually, they are not recorded in the history variable* h_N *and never become relevant.*

After receiving message c *on channel* C_1 *the mode transition system* M *returns to mode* m_1 *and the embedded mode transition system* N *becomes active again. If the history variable is not reset, the mode transition system* N *changes to mode* n_2 *in time-interval* 11 *after receiving another message* a *on channel* C_2*.* ♣

There is a special case that desires our attention. Due to strict causality any service covers at least two time intervals. The last input and first output of succeeding mode-behaviors overlap.

Without restriction let P (parent) and C (child) be two mode transition systems such that C is a mode behavior of some mode of P i.e. they are part of a hierarchic mode

transition system. A transition condition that is satisfied at time $\mathtt{t} = \#(\mathtt{i}) - 1$ in the child mode transition system C causes a special situation. According to the definition, the initially active mode behavior of C controls the outputs until time $\mathtt{t} + 1$ which is the complete output stream now. The new behavior starts only after time $\mathtt{t}$ by reading inputs.

If the time of mode change in C is $\mathtt{t}\#(\mathtt{i}) - 1$ only one time interval is left for the new mode. This last time interval's input is an input to the successor mode of the parent mode transition system P. Therefore, the mode change of the child mode transition system C happens, but the new mode will not yet become effective. It will only be effective after the parent mode transition system P switches to some mode where C is a valid mode behavior.

The inputs of the last time interval are not stored by the child mode transition system or its mode-behaviors. The mode-behavior that was active most recently in the child mode transition system C begins to read new inputs as soon as the child mode transition system C is active again. Figure 4.16 presents an example for the considered scenario.

EXAMPLE 4.11 (NESTED MODE TRANSITION SYSTEMS (CONT.))
Figure 4.16 presents an example run for the special case. The conditions $\mathtt{c}_2$ and $\mathtt{c}_3$ are satisfied together. Condition $\mathtt{c}_1$ leads to the activation of $\mathtt{m}_2$. However, the message $\mathtt{a}$ in channel $\mathtt{C}_2$ is still passed to the child mode transition system and processed there. This message leads to a transition to mode $\mathtt{n}_2$ but the reminder of the phase allows no according reactions because the parent mode transition system withdraws the control.

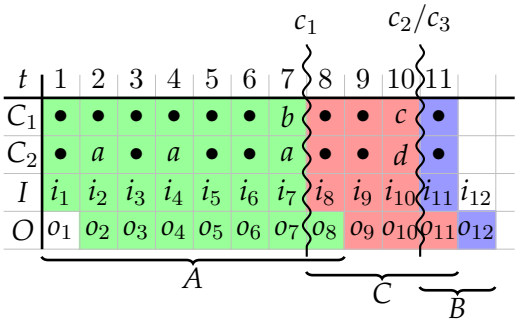


Figure 4.16.: Run of nested mode transition system with simultaneously two nested conditions becoming true requiring reactions on both levels.

Figure 4.17 illustrates the construction of the streams according to the mode transition system in Figure 4.14. The special entity $\mathtt{ANY}_1$ covers the last phase with length $\mathtt{1}$ in the child mode transition system.

To avoid yet another case in the definition to deal with this special case, we used a trick: we introduce the special semantic tuple $\texttt{ANY}_1$. This tuple covers situations where the reminder of a phase has length 1. It allows the active mode transition system to pass the control to the successor mode similar to transitions in the middle of the phase. $\texttt{ANY}_1$ makes no restrictions to the values. Its only property is that the input and output stream have length 1. Therefore, the successor mode $\mathtt{m}'$ effectively influences no outputs yet but is ready after C is activated again. Figure 4.17 illustrates the role of $\texttt{ANY}_1$.

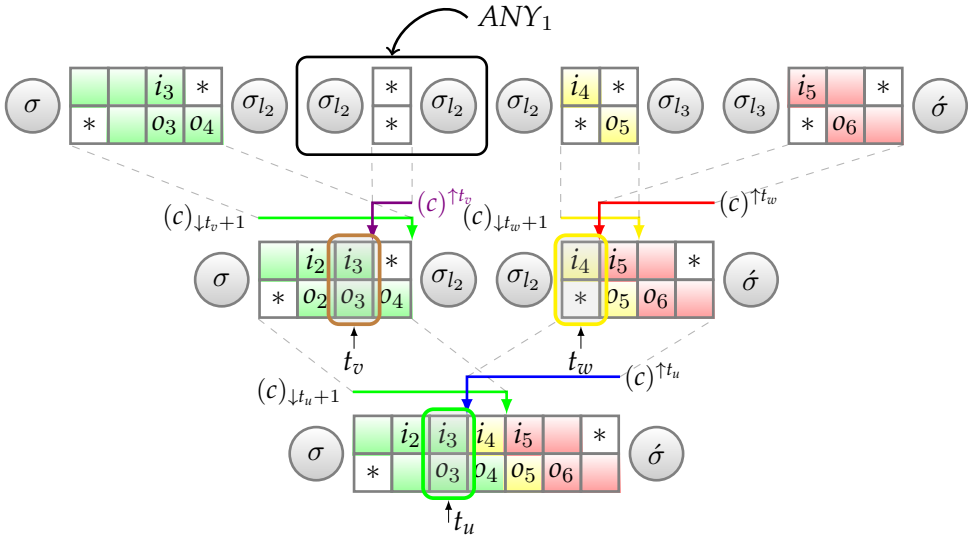


Figure 4.17.: Illustration of the construction of a stream in a hierarchic mts that includes an immediate mode transition

Underspecification of variables The semantics of alternative- and parallel composition allows arbitrary valuations for uncontrolled variables. This is an effect of underspecification of the variables. If a variable shall have a specific value, one needs to specify the value explicitly. For this reason, we made no restrictions to the set of core variables for alternative- and parallel composition of services.

The situation is different for mode transition systems. Mode transition systems separate phases of execution and relate them by mode transitions. The mode transition systems introduce specific variables that are necessary for controlling the switching behavior. Those variables have an auxiliary character and do not contribute to integrating views on a system directly as core variables in parallel or alternative compositions do.

We want to prevent tempering with those variables because this results in a hardly understandable and hardly usable formalism. In a nested mode transition system a child mode transition system's mode-variables would have arbitrary values after suspension and reactivation of the mode transition system, if we treat them as underspecified. As a result, the child mode transition system would be in an arbitrary mode. Even worse, the history variables would have arbitrary values i.e. may store any type correct stream of arbitrary length that has no relation to any earlier received messages.

If this was possible, the mode-variables would be completely unrelated to the modes and the context afterwards. Therefore, mode-variables must not be uncontrolled at any time and we require the set of mode-variables to be identical for all mode-behaviors (cf. the interface definition of mode transition system in Section 4.5.1.1).

By this we force all mode-behaviors to concern about the mode-variables existing in any other modes. The easiest setting is to define all unused mode transition system variables as unchanged explicitly, e.g., by using the service *SAME* for an adequate set of variables and composing it in parallel to the mode behaviors.

However, the formalism allows different settings like, e.g., using a different mode transition system with different transitions acting on the variables. This is useful to model scenarios where different nested mode transition systems are active in different situations but their mode decisions are relevant for each other.

Time of interruption The definition of mode transition systems forces a mode change exactly at the first time where some outgoing transitions condition is satisfied. Services are prefix closed i.e. for every behavior there is a prefix that is again a valid behavior. This allows interrupting services at any time in MARLIN (see the property of infix closure cf. Section A.2).

As a result MARLIN only supports **preemption**. The reason for this choice of interrupt handling in MARLIN is to maintain the distribution of parallel composition over mode transition systems as presented in Appendix A. We use this for some of the methodological aspects that we describe in Chapter 5.

It is possible to model weak preemption explicitly. The idea is to insert a new (intermediate) mode between a mode m and its supposed successor m'. The intermediate mode covers those actions required to bring the system to a desired state before continuing in the new mode m'. Let $F = (M, \delta, \Phi)$. We assume that a mode m shall continue until a safe state before the mode transition system follows the transition $(m, (c, l), m')$. To realize this we change F to $F' = (M', \delta', \Phi')$ by:

1. introducing an auxiliary mode m'': $M' = M \cup \{m''\}$

2. replacing the transition:

   $$\delta' = (\delta \backslash \{(m, (c, l), m')\}) \cup \{(m, (c, \rho), m''), (m'', (c'', l), m')\}.$$

   The first transition continues the history variable. The second transition treats the history variable according to the original transition

3. adapting Φ such that $\Phi(m'') = \Phi(m)$

4. the new guard c'' is satisfied if a special message at one of the output channels of the mode-behavior of m'' occurs that indicates the accomplishment of the behavior.

If no message exists that uniquely indicates the state to safely interrupt the behavior, it is an option to extend the mode-behavior of m'' with a new service by parallel composition. This new service has one dedicated output channel that serves only for this special purpose.

As a result the mode is interrupted immediately if the original condition occurs. The successor mode continues the behavior until it reaches a desired state. Only then a next mode with a different behavior becomes active.

Scope of conditions We chose to evaluate time periods in transition conditions. To be precise, a condition may cover any time period of the history until the present time. The only constraint is the decidability of the condition. Especially we restricted the conditions to regular expressions. This choice complies with the definition of context in Section 3.1.2 and allows capturing situations where a combination of events over the time indicate their beginning or ending. An example from the case study is a situation where the gesture operation is deactivated if five gestures are detected without a valid key in a row and without starting the engine in-between (cf. UC13 on Page 264).

As a special feature we introduced a mechanism that resets the history variable. This effectively allows modeling different scopes of evaluation. In the case study, e.g., the history variable is reset after the engine is started. This models the reset of the protection feature after the engine is started. Without that we need to write the condition in a way that only messages after the last engine start are relevant. In more complex mode transition systems this easily becomes infeasible and in fact is not modular any more: in a condition we need to refer to other transitions that lead to the current mode. If one of those incoming transitions is changed later we would need to check the outgoing transitions if they referred to the changed incoming transition condition and change it accordingly.

Single time interval evaluation – and hence simple contexts – are modeled by writing the condition such that it considers only the last messages at the respective channels (as an example see condition c_1 in the upcoming example 4.10).

Suspension and resumption of behavior MARLIN has a shared variable semantics i.e. variables are available for all modes and the mode-behaviors are able to manipulate them. This semantics fits to the idea of services representing different views: in different modes the system manipulates controlled resources within the scope of the projection according to different rule sets.

As an example, an adaptive cruise control is a logical function that acts on a certain state space while applying different rules for the speed control depending on a heading car. Similar, in the case study of this thesis different services manipulate the doors' lock state by applying different rules in different contexts.

Any of the other settings (reset and persistence) are reasonable as well for various reasons. Therefore, we explain how to model these settings in MARLIN. In the following description let P be the parent mode transition system and C some child mode transition system.

Continuing context observation If C is suspended after a mode change, the syntactic restriction of MARLIN ensures that any other mode-behavior of modes in P consider the mode-variables of C. If it is necessary for C to trace the contextual events even if it is suspended, an update of the history variable h_C can be made by another service.

In MARLIN we introduce a new mode-behavior that we call $LISTEN_C$ and add it by parallel composition to all those modes of the parent mode transition system where C is inactive. The service $LISTEN_C$ is simple. It has the same modes as C but no mode transitions. Furthermore, we assign *TRUE* to the mode-behaviors of all the modes. Formally, if $C = (M, \delta, \Phi)$ then

$[LISTEN_C]$

in I_C
out O_C
var $\underline{PC_C}$; $\underline{h_C}$

$LISTEN_C = (M, \varnothing, \Phi')$ where $\forall\, m \in M : \Phi'(m) = TRUE$

The result is a mode transition system that does nothing but updating the history variable. It does not restrict the resources of the parent mode-behavior nor does it affect any other mode-variables. Everything it does is to add h_C and PC_C to the set of mode-variables and updates h_C properly while leaving PC_C unchanged.

Resetting mode-behaviors Resetting a mode-behavior is to set its variables to predefined values. In MARLIN we introduce an intermediate mode that explicitly manipulates the variables accordingly. This has two benefits:

- it is possible to provide multiple sets of initial values that are chosen depending on additional conditions.
- Setting a variable to a dedicated value takes at least one time interval. Using a dedicated mode allows controlling the outputs during this time explicitly. This contrasts with, e.g., [Alur and Grosu, 2000] where the initialization of a mode always repeats the last outputs.

To implement a reset, we introduce an intermediate mode with an exit condition that becomes true after exactly one time interval. The mode-behavior of this intermediate mode is a single step that sets the variables and makes the outputs as needed, independent of the current state and inputs.

Let $F = (M, \delta, \Phi)$ be a mode transition system. Without restriction a reset shall happen if a transition $(m, (c, l), m') \in \delta$ is enabled. To realize the reset we change F to $F' = (M', \delta', \Phi')$ by:

1. introducing an auxiliary mode m'': $M' = M \cup \{m''\}$
2. replacing the transition:
   $\delta' = (\delta \backslash \{(m, (c, l), m')\}) \cup \{(m, (c, l), m''), (m'', (true, l), m')\}$.

3. adapting Φ such that $\Phi(m'') = F_{reset}$ is a service that sets the controlled resources as needed.

The transition from m'' to m' ensures that mode m'' is active for exactly one time interval. The transition condition *true* matches any input. According to the semantics of mode transition systems the transition condition evaluation starts at time $t > 0$. Therefore, at time $t = 1$ the condition *TRUE* is satisfied. The service F_{reset} sets all variables to the desired values. This includes core variables and mode-variables. In addition the service F_{reset} specifies the outputs for the single time interval.

Resuming mode-behaviors We use the already introduced service $SAME_V$ that enforces the values of variables in a next state to remain unchanged if a mode-behavior shall halt completely i.e. it shall not contribute to the systems outputs nor shall its related variables change.

4.6. Hiding, abstraction, and systems

The composition operators that we presented up to now compose more complex service out of simpler ones. Subsequently we discuss and present an operation for channel hiding and abstraction to support modularity and the transition from a service based specification of the system behavior to a component based refinement of the behavior.

4.6.1. Channel hiding

In the context of service based specifications we focus on the observations at the system interface. At this time, we shall consider no internal communication as an obligation for any implementation. We only describe the systems interactions with its environment. To enforce this view, e.g., [Harhurin, 2010] requires the output interface of one service to be disjoint with the input interface of any other service. This prevents inter-service communication. Calculations that are part of multiple services have to appear in each of them separately. This choice is a strict interpretation of defining services as projections. If projections are overlapping anything within the overlap needs to be part of all involved services.

In contrast, [Broy, 2010] uses inter service communication to give information about modes to other services. This considerably eases specifications. Repeating calculations of overlapping projections possibly involves complex calculations. Repeated writing of the calculations is cumbersome and error-prone, especially if the calculation must be changed. The use of mode communication dispenses with this redundancy. However, the communication used in [Broy, 2010] allows mutual feedback which is a characteristic of architectures and components.

For complex calculations it is a natural choice to a) decompose them such that intermediate results are handed over and b) reuse them to prevent redundancy with all its pit falls. This supports the understanding of descriptions and supports their validation. However, The "virtual architecture" that arises thereby must not be an obligation

for the actual software architecture. The "virtual architecture" takes no non-functional requirements into account that usually guide the architectural work. Therefore, we offer a hiding operation for effectively removing the structure as a proof obligation for a valid refinement.

In MARLIN we follow the lines of [Broy, 2010]. Formally we allow internal communication. However, in difference to [Broy, 2010] where the mode communication is a vital part of the specification methodology, we recommend following the paradigm of [Harhurin, 2010] and using internal communication only if it contributes to the understanding of the specification.

Parallel composition does not hide internal channels by default because we want to allow for integrating an arbitrary set of views that may affect the involved channels before hiding these channels. Hence, we offer an operator to hide those channels explicitly.

<S> ::= $\nu_{(\texttt{<chset>})}$(<S>)

Channel hiding is applicable to any set of channels and hides all channels of the set *CHSET*. The operator transforms a service into a different service whose interface is restricted to the remaining channels but retains the observations with respect to these channels.

DEFINITION 28 (SEMANTICS OF RESTRICTION):
Let A be a service with $A \in \mathbb{F}(I \rhd O, V)$ and let CHSET be a set of channels. The interface and variables of the restricted service $\nu_{(CHSET)}(A)$ is

- $I = I_A \backslash CHSET$
- $O = O_A \backslash CHSET$
- $VM = VM_A$
- $VC = VC_A$

The behavior of the restricted service is:

$$\begin{aligned}[\![\nu_{(CHSET)}(A)]\!] \stackrel{def}{=} \{(\sigma, i, o, \acute{\sigma}) \mid \exists\, i' \in \vec{I_A},\ o' \in \vec{O_A} :\\ i = i'|_I \ \wedge\ o = o'|_O \ \wedge\\ (\sigma, i', o', \acute{\sigma}) \in [\![A]\!]\ \}\end{aligned}$$ □

Hiding input channels is of no use. It just introduces additional non-determinism. Two input vectors that differ only at those channels that are subject to hiding, will appear identical after hiding. Their related outputs become valid outputs for the now identical inputs.

Nevertheless, we define the channel hiding operator more general to match the idea of slicing presented in Definition 9 and used in Sections 3.1 (definition of context aware systems) and 5.2.1.4 (relation of feature interactions and context adaptation).

Internal channels between services communicate values in a strongly causal way. The sending service uses the channels only for writing and the receiving only for reading. In the case of feedback loops, the service reads and writes the channel. In that sense, a channel realizes a local store. In contrast to variables this store already satisfies the mentioned read/write restrictions automatically. By hiding channels their characteristics become like internal variables that satisfy certain restrictions.

Despite the similarity of internal channels and variables we choose to remove hidden channels from the set of resources instead of changing their characteristic from a channel to a variable. This is to ensure the separation of

- channels as communication media between the system and the environment – with internal communication as an axillary means to enable modular specifications
- variables to express a state of a system that is possibly affected by different views that originally are expressed as different use-cases.

4.6.2. Systems

MARLIN uses states for composing services sequentially. In contrast, system specifications along the lines of [Broy and Stølen, 2001] and [Broy, 2005] focus on the I/O-behavior as a black box with channel histories replacing auxiliary states. We regard MARLIN as a means for constructing such specifications.

To transform service specifications into I/O relations $\mathbb{H}_\infty(I) \rightarrow \mathcal{P}(\mathbb{H}_\infty(O))$ where $\mathbb{H}_\infty(I)$ and $\mathbb{H}_\infty(O)$ are only infinite channel histories we apply some operations.

As a first step, we choose the initial values of the controlled resources. For a service specification F we denote this choice as a tuple (F, Λ) where $\Lambda \subseteq \mathcal{P}(\Sigma_F, O^*)$. The first element of Λ contains the initial states. The second element defines the initial outputs. We allow arbitrary combinations of initial outputs and states.

We apply an allocated variable hiding using the initial states. Note that the allocated variable hiding is no regular operator in the MARLIN-language since it does not work well with suspending and resuming services in mode transition systems.

DEFINITION 29 (ALLOCATED HIDING):
Let $F \in \mathbb{F}(I \rhd O, V)$ be a service specification. Furthermore, ς is a function $V \to \mathbb{D}$ mapping variable names to values such that for each $(r, s) \in \varsigma : s \in TYPE(r)$. Allocated hiding selects those behaviors that start in the state ς and hides the variables. We denote allocated hiding by $\tau_\varsigma(F)$.

$$[\![\tau_\varsigma(F)]\!] \stackrel{def}{=} \{(i, o) \mid \exists \acute{\sigma} : (\varsigma, i, o, \acute{\sigma}) \in [\![F]\!]\} \qquad \square$$

In a second step, we generate infinite behaviors out of the arbitrary long but still finite ones of the service. As a preliminary we assume that the service is input complete. The analyses in Chapter 6 show how to ensure this. We define a function that performs the interface abstraction.

DEFINITION 30 (INTERFACE ABSTRACTION):
The interface abstraction S of a service specification F with selected initial resources (F, Λ) is a function:

$$\Omega : [(\mathbb{F}(I \rhd O, V)) \times \mathcal{P}(\Sigma_F, \mathbb{H}_1(O))] \to [\mathbb{H}_\infty(I) \to \mathcal{P}(\mathbb{H}_\infty(O))]$$
$$\Omega(F, \Lambda) = \{(i, o) \mid \forall n \in \mathbb{N} : n > 2 \Rightarrow$$
$$((i)_{\downarrow n}, (o)_{\downarrow n}) \in \bigcup_{(\varsigma, o_1) \in \Lambda} \{(i, o) \mid (i, o) \in [\![\tau_\varsigma(F)]\!] \wedge o.1 = o_1\}\} \qquad \square$$

The interface abstraction turns a MARLIN specification into a black-box FOCUS specification for the sake of using it in an architecture as a yet unrefined black-box. The idea behind the transformation from service specifications to system specifications bases on the idea of Büchi automata [Thomas, 1990]. Services are built from finite elements and hence are finite structures. To accept infinite "words" Büchi automata require that the final states are visited infinite often. Instead of requiring some finite states that the system visits infinite often, for our purpose it is sufficient that there exists some state that the system visits infinite often.

4.7. Summary

MARLIN is a specification language that supports the initial formal specification of systems that have properties that can be treated in a context-adaptive approach.

The language is a general purpose specification language. The specification of the core system addresses all kinds of functions and MARLIN is able to provide according means. These means are not new. The specification of atomic services and their composition are concepts that are used in many other service based approaches as well (see the related work for a discussion).

The particular definitions, the semantics, and finally the selection of operators of MARLIN are tailored for context-adaptive systems and our goal of identifying and addressing insufficiencies in functional specifications:

- The binary composition operators are parallel and alternative composition and allow integrating and extending functions. These operators are mainly intended for the specification of the core system.
- The conditional composition combines different functions taking into account additional conditions. These additional conditions are explicitly modeled. Conditional composition is intended to model the adaptation subsystem.
- The semantics of MARLIN is based on stream processing functions combined with initial and final states to ease the description of exchanging the active functions and the handover of information between sequential executions of services.
- Hiding operations and abstraction allow to transfer the specification into a fully stream based semantic domain to support maximum abstraction. This is important to avoid early determination of implementation details.
- The algebraic properties of MARLIN that Appendix A presents, allow a flexibility with the order of adding behaviors and mixing conditional composition with parallel composition to address the logical grouping of functions according to the designers system understanding.

4.8. Related work

In this section we compare MARLIN with different approaches from the literature and discuss similarities and differences.

A large number of formalisms are available for the specification and model based development of systems. Among these approaches a number of fundamental different paradigms exist with a number of derivatives each tailored for different purposes. We only focus on the main representatives of the paradigms and those derivatives that relate to MARLIN directly. We organize the discussion in three sections: general formalism related aspects, adaptation/mode related aspects, and service relates aspects.

4.8.1. General purpose formalisms

To relate MARLIN to general purpose approaches and formalisms, we mainly address a couple of properties. These properties are a) the intended use in the life-cycle and b) the semantic domain that is used. Aspects of the semantic domain are among others the communication paradigm (synchronous vs asynchronous) and the semantic class (e.g., operational semantics vs. denotational semantics). Both aspects have some influence on, e.g., conflicts that may arise during compositions, modularity of properties and the applicable proof concepts.

Process algebras. Process algebras are formalisms that deal with algebraic properties. Well known representatives are Communicating Sequential Processes (CSP)

[Hoare, 1985], the Calculus of Communicating Systems (CCS) [Milner, 1982], the algebra of communicating systems [Bergstra and Klop, 1985] and – especially for mobile systems – the π-calculus [Milner, 1999]. All of them define an algebraic structure and define systems as sets of terms. This idea appears in MARLIN as well. Service expressions are manipulated along the lines of the algebraic properties.

Process algebras use processes to describe interactions. A process represents a system in a (control)state. Starting in this state a number of interactions are possible which the process terms define. The set of process terms finally can be regarded as a definition for a state transition system. This aspect is similar to MARLIN. The service terms (at least their syntactic representation) define a state transition system as well. Both approaches use states (or processes) to combine behavioral entities (services in MARLIN and processes in the process algebras).

The actual usage of states differs between process algebras and MARLIN. MARLIN uses states as containers for data and allows access to the variables. Process algebras use the "process" as a representative for a system's capabilities to interact and hide possible dependencies on data in different "processes". The processes encode a possible data state.

The various process algebras differ in the semantics they assign to the process terms. Depending on the semantic domain the states play a more or less important role in the semantic mapping. For operational semantics the states are an inherent part whereas denotational semantics removes them a far as possible. Bergstra shows that operational semantics and denotational semantics can be applied to (almost) any process algebra [Bergstra et al., 1988]. MARLIN uses a denotational semantics based on streams that we extended by data states to enable the sequential extension of behaviors.

In contrast to process algebras MARLIN has no focus on internal communication. In contrast to process algebras that use communication to link independent entities of behavior, we intend (inter service) communication in MARLIN only for shortening specifications and avoiding redundant expressions. Therefore, MARLIN provides no sophisticated communication model. Information possibly flows via common variables and inter service communication. Similar to the Janus approach of Broy [Broy, 2005], MARLIN uses asynchronous communication and strict causality. Most process algebras use synchronous communication although there are efforts to offer an asynchronous semantics like presented in [de Boer et al., 1992].

While the various semantics for process algebras put a focus on compositionality, MARLIN has its focus on overlapping views. This in an important difference. Compositionality is necessary to allow replacing some entity (possibly a component) by another one with the same observable behavior but having a different structure or implementation internally. This is of no use in the context of MARLIN. Since a service and the inter service communication shall be no obligation for any implementation, it is no considerable scenario to replace a service by another one that behaves equally but has a different structure or implementation. The only reason to replace a service is if the new one has a new behavior or affects different resources.

However, the semantics of MARLIN is compositional but needs the variables to be included in the semantics. For process algebras (and other component approaches) this

is considered as to less abstract because it would restrict possible implementations. For MARLIN as a service based language this does not matter.

Masaccio. The formalism Masaccio [Henzinger, 2000] is relates to MARLIN because it has similarities to the formalism presented by Schätz [Schätz, 2006, 2007, 2009] which in turn has much influence on the style of specifying the core system in MARLIN. Masaccio originally is a component specification technique that allows the parallel and serial composition of components. Similar to MARLIN, Masaccio considers finite sequences as executions. The sequences are defined over a set of variables.

The serial composition is comparable to the alternative composition in MARLIN. However, in a serial composition Masaccio requires components to agree on the output interface to prevent additional non-determinism by unrestricted output channels. Any composition relies on so called control points. These control points are the meeting points for behaviors where they start and end. In contrast to MARLIN control points are obligatory because they define possible splices for serial composition. Between the splices a behavior cannot be interrupted. In MARLIN we emphasized the possibility to interrupt a behavior at any time to react to changes in the context and to preserve distributivity.

Hiding variables in Masaccio results in local variables. During resuming a behavior in Masaccio local variables are initialized non-deterministically. It is impossible to save the value of local variables during an interruption. In MARLIN we renounced hiding of variables. We are only interested in the interface behavior and use variables only to save states during mode changes.

The differences between Masaccio and MARLIN are mainly consequences of their different applications. MARLIN focuses on interface specifications and Masaccio focuses on component architectures. The composition in Masaccio needs to preserve input permissiveness and deadlock freedom. Services in contrast can be partial and allow for a less restrictive composition. Furthermore, Masaccio offers no explicit means to capture adaptive behavior.

Services by Schätz. The formalism of Schätz [Schätz, 2006, 2007, 2009] is closer to MARLIN because it generalizes the composition operators to services. The way Schätz defines the composition operators is the paragon for the specification of the core system in MARLIN. However, Schätz still adheres to the principle of explicit control locations. Furthermore, he provides no explicit mode concept. Switches between services only happen at associated entry and exit points. Arbitrary switches at any time as we require them for context adaptation is unreasonable in this formalism because it would need to associate all control locations with each other.

Schätz gives a semantics to transitions from entry to exit points as sequences of interactions together with all prefixes. By hiding control points the sequences of interactions become longer. This is different to composition operators in MARLIN which automatically create all sequences of behaviors of any length. The longer behaviors in Schätz's

approach are no longer interruptible because they lack the required intermediate control location

The Janus approach. The semantics of MARLIN uses streams of messages. This goes back to ideas of Kahn's "natural semantics" [Kahn, 1988] sometimes also called a "big step semantics" [Leroy, 2010]. We use the term "big step semantics" because it fits better to MARLIN which links two states by a transition that possibly involves arbitrary interactions (a big step). However, the usage of streams in MARLIN is mainly influenced by the FOCUS theory [Broy and Stølen, 2001] and the JANUS approach [Broy, 2005, 2010]. Both use infinite timed streams as the semantic domain.

MARLIN uses finite timed streams to allow sequential execution of services. In contrast to JANUS, the explicit modeling of modes in MARLIN requires finite behaviors and means to define interruption and resumption of behaviors. This allows reasoning about phases of execution and the conditions of mode changes. We changed the original concept of a pure stream semantics for this purpose. Furthermore, we are interested in the analysis and handling of conflicts. This requires means for supplementing behaviors like the alternative composition in MARLIN.

4.8.2. Adaptation and modes

To relate MARLIN to approaches that focus on adaptation or similar concepts we focus on the aspects that are specific for interrupting and continuing behaviors and the possibilities to define contexts and the context switches.

Statecharts. The Statecharts formalism [Harel, 1987] is strongly related to modes. Von der Beeck presents an overview of different semantics for Statecharts and the related problems [von der Beeck, 1994]. Therefore, we refer to Mini-Statecharts [Nazareth et al., 1996b,a] because they offer a compositional semantics that is close to the semantics we chose for mode transition systems.

Hierarchically decomposing states is the most useful property in Statecharts for specifying context-adaptive systems. Any state possibly contains sub-states which are related by state transitions. The original Statecharts allowed inter level transitions which prevent a compositional semantics. Therefore, the Mini-Statecharts restrict themselves to transitions at the same level. This concept resembles the mode transition systems in MARLIN. Statecharts in general show a rigid hierarchy of states.

Distributivity of parallel composition over hierarchical states is not considered although possible. It is simply out of scope of Statecharts since parallel composition composes components (representing modularized behaviors) and distribution is transverse to this concept. In contrast, distributivity is an important property in MARLIN and supports the generation of normal forms for the analysis of certain properties (cf. Section 6.2.2).

Similar to Mini-Statecharts, MARLIN continues a mode-behavior until a superordinate mode transition system executes a transition. In Mini-Statecharts a stream

semantics is only available for the complete system. The definitions of composition operations use a single step semantics. MARLIN in contrast uses the stream semantics throughout the formalism and provides a single step semantics only as an additional means to ease certain analyses (c.f. Section 6.3.3).

Extensions of Masaccio. Based on the work of Henzinger, Alur et al present an extension of Masaccio with a hierarchical concept similar to modes [Alur and Grosu, 2000]. The modes allow a history retention if needed. Default entry and exit points and a history variable similar to Statecharts exist for this purpose. A mode is a kind of refined state. However, mode changes are only possible if a mode reaches an exit point. This is in contrast to MARLIN and many flavors of state charts and only supports weak preemption. As a result, this prevents interrupting behaviors at arbitrary times and a preemptive reaction to changes in the environment.

We regard a hierarchy as a concept where decisions made at a higher level in the hierarchy govern decisions at lower levels. A hierarchy is a method to structure aspects and views of systems. Hiding lower levels of a hierarchy needs to sustain the possibility to understand aspects of higher levels. In the formalism of Alur the behavior of higher levels depends on that of lower levels because transitions at higher levels cannot be taken if the lower levels disallow them. For our purpose this prevents the usage of Alur's approach for specifying context-adaptive systems as we regard them.

Extensions of Janus. In a recent publication, Broy presents an extension to the JANUS approach that explicitly deals with modes [Broy, 2010]. His goal is to address dependencies between services that are a consequence of overlapping projections. Inter service communication propagates mode information. An influencing service sends information about its mode and influenced services may receive this information and react accordingly.

In Broy's approach modes are only implicit. All services are still composed in parallel. The dependency on modes appears in all influenced services as an additional column in the tabular representation of the service specification. There is no focus on the separate specification of services and modes and no concept of hierarchical modes is available. This is contrary to our goal of specifying the behavior of context-adaptive systems by isolated services each responsible for different contexts and composing them using additional information about contexts. From a formal point of view, adding rows to tables in Broy's approach roughly correspond to alternative composition in MARLIN. However, in Broy's approach only complete tables have a semantics without the option to supplement the tables later by a composition operator.

Services by Harhurin. Harhurin [Harhurin, 2010] describes in his PHD-thesis a method for the service based specification of systems. He observes dependencies between services just like Broy but resolves them with a priority system. Therefore, he introduces a composition operation that uses a priority automaton which defines the

activation of either of two (possibly conflicting) services. If one service has priority over another one, the less prioritized one is suspended. The priority automaton roughly corresponds to our mode transition systems. However, the mode automaton only considers single events. Complex structures of modes and mode changes are not modeled easily because the mode automaton only composes two services and captures their (de)activation.

A more formal difference to the approach of Harhurin is that services only share channels but no variables. The sharing of channels includes input and output channels but excludes internal communication. In MARLIN we follow this pattern from a methodological point of view but offer means for internal communication for the reasons that we described in Section 4.6.1. Furthermore, disallowing shared variables is contrary to our goal of integrating different views with shared variables indicating commonalities between the views.

The formalism of Harhurin always resumes reactivated services. MARLIN provides a more flexible mechanism allowing resuming, resetting, or continuing a behavior at states that other services updated meanwhile. Harhurin's approach offers no alternative composition. The priority system is the only way to resolve conflicts without extending behaviors.

Mini-Lustre Maraninchi et al. [Maraninchi and Rémond, 1998] describe ideas about modes that have influenced the use of modes in MARLIN. They regard modes as an orthogonal concept and describe behaviors sequentially instead of concurrently. Similar to MARLIN they regard a system specification as a single, huge state transition system with fully detailed state space. Since such a specification is not manageable for large systems they use modes to structure this specification.

Maraninchi et al. suggest a mode concept based on Lustre [Caspi et al., 1987] which they call mini-Lustre. Mode automata relate mini-Lustre programs with each mode. Mode changes are preemptive. Mini-Lustre is a high level programming language like notation. Therefore, the nature of specifications in mini-Lustre is like structured programs and not like high level specifications. Maraninchi et al. give a trace semantics to mini-Lustre programs without mapping the mode concept into the semantic domain. They translate the mode concept into a mini-Lustre program using if ... then ... else constructs before giving the programs a semantics.

Organic computing approaches In a series of publications [Gudemann et al., 2008; Nafz et al., 2009, 2010] Nafz et al. describe the development and proof principles for organic computing systems. They introduce the so-called organic design pattern which organizes a number of agents with different capabilities. According to the agents' capabilities they take roles. Roles define the processing of resources and the required capabilities for the processing. Reconfiguration reassigns roles if, e.g., agents loose capabilities and are no longer able to fulfill their roles.

Treating components as roles, this approach allows a dynamic realization of the components by agents like robots or control units in the automotive domain. The approach

supports the self-X properties [Kephart and Chess, 2003] that describe a system's ability to adhere to given functions in an environment with changing resources. The systems behavior is a sequence of states of which some are productive i.e. the system behaves as intended, and others are unintended i.e. the system has to reconfigure to return to a productive state. The method is called Restore-Invariant-Approach (RIA) because productive states are characterized by invariants. A constraint solver working on relational logic finds valid configurations.

In contrast to MARLIN the approach aims at finding configurations at run-time. This works well in a setting with interacting agents with dedicated capabilities without automatically determining different functions for different contexts. This approaches the dynamic realization of functionality and is relevant for later stages of the development (i.e. defining the architecture) and covers dynamic reconfiguration.

Software Cost reduction (SCR) The Software Cost Reduction Method (SCR) [Heitmeyer et al., 1996] uses a similar view on systems than we do. Developed for the easy formalization of requirements, SCR considers only inputs, outputs, and possibly a local state of the system. The specification implies no structure and no realization details. SCR uses table techniques for specifying and maps the tables to an automaton model giving tables an operational semantics through state transitions. Composition in SCR is limited to the use of functions in tables which defines a term substitution system. The aspects of the tables in SCR and the projections of MARLIN are different. MARLIN allows the specification of overlapping aspects of a function always relating inputs to outputs while tables in SCR may define the calculation of intermediate results.

The mode concept of SCR differs from that in MARLIN, too. In MARLIN, modes describe the reactions of a system as long as a certain context is effective. In contrast, modes in SCR have characteristics of states of the system that influence the possible calculations. Each table whose function depends on a mode must include the mode into the calculations. In MARLIN, the functions are independent of modes they appear in and in fact can appear in different modes.

Message sequence charts Message sequence charts (MSCs) capture interactions and qualify for modeling and composing use-cases. Krüger presents a formal approach for combining MSCs [Krüger, 2000]. In his approach, Krüger presents a number of composition operators for MSCs. One is a guarded combination and allows selecting behaviors depending on a given predicate. The predicate's interpretation is limited to the state space and allows only indirectly a choice, depending on prior inputs that manipulate the state space. Modeling of dependent behavior is more implicit without the clear separation of context and core behavior. Furthermore, many composition operators already aim at defining architectures. This matches to the idea of using MSCs to describe the interactions between components as a refinement of using MSCs to capture the interactions between the system and its environment.

Chapter 5

Methodological aspects of MARLIN

Services are a formal representation of possible interactions with the system i.e. use-cases of the system. The goal of composing services is creating a single system specification out of separated use-case descriptions. In the context of "context-adaptive systems" the integration must account for contextual conditions as well. This especially includes the systematic treatment of overlapping contexts and the handling of the related services.

In this chapter we focus on the methodological aspects of applying MARLIN to specify context-adaptive systems. We introduce no new method but discuss aspects of context-adaptive systems and their application in the context of existing methods like, e.g., described in [Rittmann, 2008].

The methodological considerations start in Section 5.1 with extending contextual requirements chunks [Sitou, 2009] as the first artifact informally describing the context-adaptive systems' requirements in a scenario-driven manner. In Section 5.2 we discuss possible relations between services that influence their integration and demonstrate that context adaptation is yet another instance of such relations.

The treatment of the additional conditions in the contextual requirements chunks presented in Section 5.3 prepares the transformation of the contextual requirements chunks into a service hierarchy. This requires adapting the definition of service hierarchies from [Broy, 2005] and [Broy, 2010] to the structure of MARLIN.

Section 5.5 shows how the service oriented approach integrates with a development life-cycle and especially with component based system architecture specifications. Section 5.6 finally discusses related work.

Contents

5.1. Preliminaries

The main focus of MARLIN is on context-adaptive systems but the language and methods are more general. MARLIN fits well with any system whose specification can be organized in modes. Therefore, we start the considerations by discussing the term context in the face of services and feature interactions. Furthermore, we describe the contextual requirements chunks of Sitou [Sitou, 2009] together with some enhancements that we need for the aspired method extension.

5.1.1. Services and context

In this chapter we describe the creation of black box specifications by integrating separate viewpoints that describe interactions with the system possibly in different situations. We use services as formalizations of these interactions. Services in turn can be composed of sub-services. The services may overlap according to the view they represent or they control the interactions in different contexts. This introduces a functional structure and the need to consider interrelations between services.

In Section 3.1.2 we used "users" and their direct input channels for defining context adaptation. The definitions refer to a system under construction as a whole without concerning about its composition from individual interaction patterns.

Projections on the interface relate services and their sub-services. This possibly moves input channels from the direct inputs of the service to indirect inputs of some sub-services. From the perspective of sub-services, information on some channels disappears in calculations. These channels merely influence the nature of the calculations indirectly.

The setting resembles the definition of context adaptation with the (sub-)service and its interactions in the center of attention. We possibly just change the interface to the

reference system: it happens to be only a subset of the user interface or some other clipping of the interface. As a consequence, the indirect informations' source can be the user, the context of the system under construction or some other service of the system.

Figure 5.1 illustrates the different roles of inputs with respect to services and sub-services. The service F with interface $(\{i_1, i_2, i_c\} \triangleright \{o_1, o_2\})$ is a mode transition system that is sensitive to inputs on channel i_c. The mode behaviors A and B have the interface $(\{i_1, i_2\} \triangleright \{o_1, o_2\})$ each.

For the service F the input channel i_c is a direct input. In contrast, A and B use no information of channel i_c. However, information on i_c controls their activation. From the viewpoint of the sub-services, the input i_c is indirect. This consideration complies with the discussion in Section 3.1 where we argued that context-adaptive systems are a matter of the viewpoint.

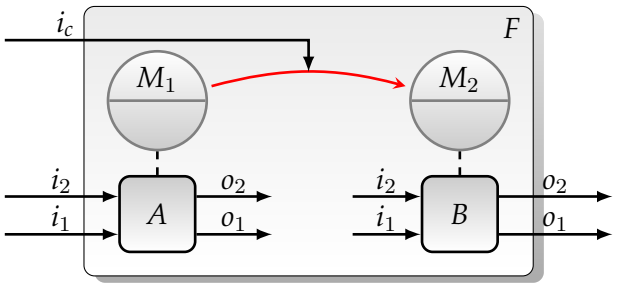


Figure 5.1.: Input channel i_c is an indirect input to the mode behaviors but a direct input to the compound service

5.1.2. Contextual requirements chunks

We assume that a first iteration of a requirements elicitation phase results in a set of requirements and associated contexts. We make no assumptions about the maturity (consistency and completeness) of the informal requirements specification. One option to organize the requirements and their contexts are *contextual requirements chunks* [Sitou and Spanfelner, 2007; Sitou, 2009]. Table 5.2 shows examples for contextual requirements chunks that are already extended according to the upcoming considerations.

Each row in the contextual requirements chunks has two aspects that we use for relating the requirements. The first aspect is the (textual) requirement itself. Written as a scenario it represents a pattern for interacting with the system. The second aspect is the context for the requirement. The original contextual requirements chunks of Sitou use simple contexts i.e. the requirement must be satisfied as long as the context condition is true. We change this to match the definition of contexts in Section 3.1 with entry and exit conditions.

The contexts in Table 5.2 are simple contexts and we may (syntactically) substitute the entry and exit conditions in the table, e.g., by $\sharp(V = hi)$ for requirement 1.

Formally, contextual requirements chunks are lists of tuples. The tuples have a name, a context, an informal description of the behavior (e.g., given as an interaction diagram)

Number	Context		Requirement
	IN	OUT	
1	$V = hi$	$\neg context_{IN}(1)$	Doors become locked, doors cannot be unlocked or opened from outside
2	$V = lo \wedge UBat = hi \wedge KPos = t \wedge$	$\neg context_{IN}(2)$	Trunk is opened on gesture (with respect to. Disturbance Protection)
3	$V = lo \wedge UBat = hi$	$\neg context_{IN}(3)$	Trunk is closed on gesture

Figure 5.2.: Clipping of the contextual requirements chunks of the keyless entry case study

and an illustrating scenario. Subsequently, we drop the illustrating scenario. It is useful for the process of requirements elicitation but we make no use of it.

Let C_1 and C_2 be two sets of channels and *STRING* the set of all strings. In the sequel we treat contextual requirements chunks (CRCs) as a relation

$$CRC \subseteq NAME \times ((\vec{C_1} \to \mathbb{B}) \times (\vec{C_2} \to \mathbb{B})) \times STRING \times STRING$$

We use the names in the first row of the list of contextual requirements chunks as names for services and define the mappings:

$$context_{IN} : NAME \to (\vec{C_1} \to \mathbb{B});\ context_{IN}(n) = in \Leftrightarrow (n, (in, out), bh, sc) \in CRC$$
$$context_{OUT} : NAME \to (\vec{C_2} \to \mathbb{B});\ context_{OUT}(n) = out \Leftrightarrow (n, (in, out), bh, sc) \in CRC$$

The mappings allow accessing the contexts of the requirements.

An active requirement in a context does not imply that the service that represents the requirement is able to react to *all* possible inputs. The requirement, and hence the respective service may be partial. This is a subject to the later analysis of the specification.

5.2. Relations between services

Specifying systems involves integrating different use-cases. These use-cases usually do not exist in isolation. They are related, describe overlapping aspects of a system and - at least in an early state of the development - may be conflicting. These conflicts are a consequence of complexity, yet unaligned requirements of stakeholders, and undiscovered special cases. Services allow the formalization and integration of use-cases without adjusting and filtering them. Through analysis of the service specification conflicts are revealed and the requirements and their relations become better known to address conflicts and underspecification systematically in follow-up iterations.

5.2.1. Horizontal relationships (feature interactions)

In contrast to self-contained components, services are strongly related. The composition of services influences their interaction capabilities by adding or removing interactions. This is caused by the overlapping of services and their partial characteristic. Following Zave, we call these relations *feature interactions* [Zave, 1993]. The literature (e.g., [Zave, 1993; Zave and Jackson, 2000; Deubler, 2008]) discusses different kinds of feature interactions.

5.2.1.1. Definition of feature interaction

Feature interactions exist between influencing services and influenced services [Rittmann, 2008] possibly in an m:n manner. Formally, services correspond to projections on a subset of a system's resources (inputs, outputs, and variables). The interfaces of the two services with a feature interaction are different but overlapping.

Abstracting feature interactions to the level of a black box we observe outputs of the influenced service depending on inputs of the influencing service. Formally, the I/O-relation of the influenced service is less deterministic with respect to its interface. The actual outputs are additionally determined by the state and inputs of the influencing service.

EXAMPLE 5.1 (INTEGRATION OF RADIO AND TELEPHONE)
Examples are a radio and a telephone in a car. Both serve different goals of the driver but control a shared resource: the audio output channel. Their integration introduces a possible feature interaction. If a call comes in while the radio plays, the phone takes the control over the audio channel. A projection to the radio only shows a non-deterministic control of the audio channel. Only the additional inputs of the phone determine if the radio has control over the audio channel. ♣

We capture this observation in the definition of faithful slices according to Broy [Broy, 2010].

DEFINITION 31 (FAITHFUL SLICE):
Let $(I' \triangleright O', V') \subseteq (I \triangleright O, V)$ *be two interfaces. We say that the projection of a service* $S \in \mathbb{F}(I \triangleright O, V)$ *to the sub-service* $S' = S\dagger(I' \triangleright O', V')$ *is faithful iff:*

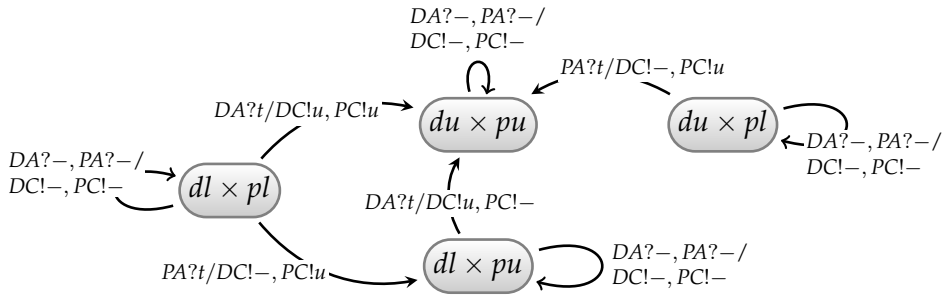
$$\forall(\sigma, x) \in dom(S) : \tag{5.1}$$

$$\{(y_{\lfloor O'}, \acute{\sigma}_{\lfloor V'}) \mid (y, \acute{\sigma}) = S.(\sigma, x)\} = \{(y, \acute{\sigma}) \mid (y, \acute{\sigma}) = S'.(\sigma_{\lfloor V'}, x_{\lfloor I'})\} \tag{5.2}$$

Informally spoken, the inputs of S' *fully determine the outputs with respect to the behavior of the system* S *(the projection introduces no additional non-determinism). We write* $faithful(S\dagger(I' \triangleright O', V'))$ □

[*OpenFrontDoors*]

in $DA : \{t\}, PA : \{t\}$
out $DC : \{u\}, PC : \{u\}$
var $DDS : \{dl, du\}, PDS : \{pl, pu\}$



In a faithful slice, the outputs only depend on the regarded inputs and variables. We say that a service $F_1 \in \mathbb{F}(I' \rhd O', V')$ is subject to feature interaction (in a superordinate service $F \in \mathbb{F}(I \rhd O, V)$ with *F super* F_1) if $F_1 = F\dagger(I' \rhd O', V')$ is not faithful.

This definition of feature interactions is descriptive and relates to the behavior of the compound service. Furthermore, feature interaction is no symmetric relation. For any number of services $F_1, F_2, \ldots$, the faithfulness of one service does not necessarily imply the faithfulness of any other service, even if two services are supplementary [Broy, 2005].

EXAMPLE 5.2 (FEATURE INTERACTION)
The example is a clipping from the case study: opening the front doors. Although small, the example exhibits the effect in question.

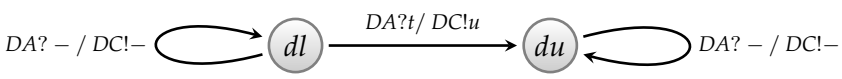
The service has two input channels: `PA` *and* `DA`*. We ignore other aspects like the global lock notification* `DL` *and focus on opening the car via the driver door trigger or the passenger door trigger. We capture the aspired behavior in the specification* `OpenFrontDoors`*.*

We investigate two sub-services describing the behavior of the driver door and the passenger door respectively. The services correspond to projections on the respective resources. The passenger door has channels `PA` *and* `PC` *and a local variable* `PDS` *and the driver door has channels* `DA` *and* `DC` *and a local variable* `DDS`*. The specifications* `ProjectionOpenDriverDoor` *and* `ProjectionOpenPassengerDoor` *show the two services.*

The projection `ProjectionOpenPassengerDoor` *is non-deterministic. Furthermore, the parallel composition of* `ProjectionOpenPassengerDoorin` *and* `ProjectionOpenDriverDoorin` *does not result in the original behavior either. The information about the actual nature of the feature interaction between the driver door and the passenger door is lost in the projection. We investigate this in the next section.* ♣

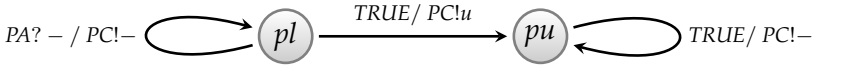
[*ProjectionOpenDriverDoor*]

in $DA : \{t\}$
out $DC : \{u\}$
var $DDS : \{dl, du\}$



[*ProjectionOpenPassengerDoor*]

in $PA : \{t\}$
out $PC : \{u\}$
var $PDS : \{pl, pu\}$



5.2.1.2. Discussing feature interactions

The definition of a faithful slice is descriptive. It allows defining feature interactions by the relation between a system's (or a service's) behavior and a slice. The definition allows identifying influenced services but gives no information about the influencing service or the nature of the influence.

Extending projections to all relevant inputs solves feature interactions (in Example 5.2, we need to include the inputs of the driver door to the projection of the passenger door). This inflates services such that they finally become faithful. However, this inflation is only reasonable for small interfaces. Furthermore, it requires the a priori consideration of any possible inputs.

EXAMPLE 5.3 (RADIO AND TELEPHONE AS FAITHFUL SLICES)
Again we use the example of the radio and the telephone. Remember that services formalize use-cases. Methodologically, one wants to formalize the original use-cases of the radio without already considering the telephone, or any other service that possibly uses the audio channel as well.

Services shall offer an elevated level of reuse compared to components because they focus on the function only. Considering all possible influences on the service during specification to make it faithful is counterproductive: in a reuse scenario, a service may be combined with different services that have different interfaces. Think of a navigation system in a later product that uses the audio output as well.

From a methodical point of view, the radio, the telephone, and the navigation system shall be specified independently. Only while combining them, possible feature interactions shall be revealed and treated. ♣

[*OpenDriverDoorV2*]

in $DA : \{t\}$
out $DC : \{u\}, PC : \{u\}$
var $DDS : \{dl, du\}, PDS : \{pl, pu\}$

DDS	*PDS*	*DA*	*DC*	*PC*	*DDS*	*PDS*
dl	*	-	-	*	dl	*
dl	*	t	u	u	du	pu
du	*	-	-	*	du	*

[*OpenPassengerDoorV2*]

in $PA : \{t\}$
out $PC : \{u\}$
var $PDS : \{pl, pu\}$

PDS	*PA*	*PC*	*PDS*
pl	-	-	pl
pl	t	u	pu
pu	-	-	pu

In a constructive approach like MARLIN the goal is integrating formalizations of use-cases which have a fixed set of input- and output channels initially rather than producing faithful slices. Therefore, we aim at a more constructive handling of feature interactions that allows us integrating the original use-cases and to analyze them with the goal to finally model the feature interactions explicitly.

We start with use-cases addressing intended behaviors. The use-cases are conflicting if their composition as services requires different values for common resources like controlled channels or controlled variables (at least for some inputs).

EXAMPLE 5.4 (FEATURE INTERACTION II)
Again we use the behavior of the service `OpenFrontDoorsV1` *in Example 5.2. This time we build up the functionality in a constructive way. We specify the behavior of the driver door (*`OpenDriverDoorV2`*) and the passenger door (*`OpenPassengerDoorV2`*) separately and ignore any other influences from other doors. We accept conflicts during their composition.*

Modeling the driver door service we capture the interaction where a driver door trigger signal is present and both, driver door and passenger door open. We model the passenger door in a similar way but focus only on the passenger door trigger as an input. The composition of the two services is unable to react to the driver door's trigger signal. ♣

In MARLIN there are two ways to override conflicts.

1. We use a mode transition system to switch the mode of the passenger door to allow a different service to open it if a driver door trigger signal arrives.

2. We model the feature interaction between the two services explicitly as a new service. We compose the new service by alternative composition to control the behavior during the conflict.

Using mode transition systems is heavy weighted because a proper mode transition system together with appropriate mode behaviors is necessary. Subsequently we describe the second approach.

[*DDPDFI*]

in $DA : \{t\}, PA : \{t\}$
out $DC : \{u\} PC : \{u\}$
var $DDS : \{dl, du\}, PDS : \{pl, pu\}$

DDS	*PDS*	*DA*	*PA*	*DC*	*PC*	*DDS*	*PDS*
dl	pl	t	-	u	u	du	pu
dl	pu	t	*	u	-	du	pu

EXAMPLE 5.5 (FEATURE INTERACTION II CONT.)
We create the service `OpenFrontDoorsV1` *by the composition*

$$\texttt{OpenFrontDoorsV1} = (\texttt{OpenDriverDoorV2} \otimes \texttt{OpenPassengerDoorV2}) \oplus \texttt{DDPDFI}$$

where `DDPDFI` *models the feature interaction.*

The service DDPDFI only handles the driver door trigger signal and overrides the conflict. The first line addresses the setting with both doors initially locked. The second line addresses the setting with an unlocked passenger door. This state is unreachable in `OpenFrontDoors` *(assuming that in the initial state all doors are locked) but is reachable in a general setting if we consider closing doors.* ♣

After mitigating the conflict, the projections to the original services' interfaces are no longer faithful. Even more, the projections to any of the original services interfaces may not be equal to one of the original service definitions. The original services are only partly observable in the final system.

From a methodical point of view, feature interactions are interesting because they relate services that capture different functions that serve different goals of the users. The involved services model different and distinguishable functions at system level.

5.2.1.3. Other causes for conflicts

Faithful slices cover only some of the relations between overlapping services: Only those that change the determinism with respect to different projections. There are other causes for conflicts that we do not call feature interactions. To address the relations between services as formalizations of use-cases, compound services, and the system we introduce additional terms.

Service interference is a generalization of feature interactions. It describes a setting where some interactions with a service are not available if the service is composed with another service.

DEFINITION 32 (SERVICE INTERFERENCE):
Let there be services A with interface $(I_A \rhd O_A, V_A)$ and B with interface $(I_B \rhd O_B, V_B)$ and $C = A \otimes B$. Without restrictions we say that there is a service interference from A to B iff

$$\exists(\sigma, x) \in \Sigma_C \times I_C : (\sigma_{\lfloor V_B}, x_{\lfloor I_B}) \in dom(B) \wedge \\ \{(y_{\lfloor V_B}, \acute{\sigma}_{\lfloor O_B}) \mid (y, \acute{\sigma}) \in C.(\sigma, x)\} \subset B.(\sigma_{\lfloor V_B}, x_{\lfloor I_B}) \qquad \square$$

A service A interfering with a service B disallows some valuations of common resources that B allows in isolation. As a result the corresponding behavior of B does not exist in the compound service. Note that this does not imply that the compound service is partial. If B is non-deterministic, the compound service still may be able to react to the concerned inputs.

DEFINITION 33 (CONFLICT BETWEEN SERVICES):
Let there be services A with interface $(I_A \rhd O_A, V_A)$ and B with interface $(I_B \rhd O_B, V_B)$ and $C = A \otimes B$. If a service interference removes all reactions to some input we call it a conflict.

$$\exists(\sigma, x) \in \Sigma_C \times I_C : (\sigma_{\lfloor V_B}, x_{\lfloor I_B}) \in dom(B) \wedge \\ \{(y_{\lfloor V_B}, \acute{\sigma}_{\lfloor O_B}) \mid (y, \acute{\sigma}) \in C.(\sigma, x)\} = \varnothing \qquad \square$$

The two definitions generalize feature interactions to composition scenarios with the involved services having arbitrary interfaces (including identical interfaces). We shortly summarize the possible relations between services A and B:

1.	Service A and B have overlapping interfaces	the composition of A and B reduces nondeterminism but does not introduce partiality	the services are in feature interaction
2.		the composition introduces (additional) partiality	the services are in feature interaction and in conflict
3.	Service A and B have identical interfaces	the composition of A and B reduces nondeterminism but does not introduce partiality	service A and B interfere
4.		the composition introduces (additional) partiality	the services interfere and conflict

If two interfering or conflicting services have overlapping but not identical interfaces, a feature interaction can be observed in terms of non-faithful slices. If the services

have congruent interfaces, interference or conflicts cannot be observed as non-faithful slices any more.

DEFINITION 34 (FAITHFUL SERVICE):
Let F be a service with interface $(I' \triangleright O', V')$ that formalizes a use-case and S be a compound service with interface $(I \triangleright O, V)$ that offers the service F, and $(I' \triangleright O', V') \subseteq (I \triangleright O, V)$. We say that service F is faithful in S iff:

$$\forall(\sigma', x') \in dom(F)\ \exists(\sigma, x) \in dom(S) : (\sigma_{\lfloor V'}, x_{\lfloor I'}) = (\sigma', x')$$
$$\{(y, \acute{\sigma}) \mid (y, \acute{\sigma}) = F.(\sigma_{\lfloor V'}, x_{\lfloor I'})\} = \{(y_{\lfloor O'}, \acute{\sigma}_{\lfloor V'}) \mid (y, \acute{\sigma}) = S.(\sigma, x)\}$$

The system offers all behaviors of F at the respective interface – maybe more but never less. We write $faithful_S(F)$ □

A service can only be faithful in a system if there are either no service interferences or all service interferences were resolved in favor of this service. After solving conflicts the system is again able to react to inputs but possibly in a different way. Therefore, we define the consistent integration of a service.

DEFINITION 35 (CONSISTENT INTEGRATION OF A SERVICE):
Let F be a service with interface $(I' \triangleright O', V')$ that formalizes a use-case and S be a compound service with interface $(I \triangleright O, V)$ that offers the service F with $(I' \triangleright O', V') \subseteq (I \triangleright O, V)$. We say that service F is consistently integrated in S iff:

$$faithful_S(A) \vee \{(\sigma_{\lfloor V_A}, x_{\lfloor I_A}) \mid (\sigma, x) \in dom(S)\} = dom(A)$$

The system either behaves like F or it provides alternatives for some inputs. We write $consistent_S(F)$ □

Of course the final specification needs to address all conflicts. The use of MARLIN is deriving the exact outcome of composing services to analyze them. This reveals possible conflicts (Section 6.3.1 presents the analysis of specifications).

5.2.1.4. Feature interactions and context adaptation

Considering systems as context-adaptive is motivated by structuring the behavior such that behaviors in different contexts can be described apart. The idea is to separate the specification of services in the core system from the logic that defines the activation of those core system services, i.e., the adaptation logic. This separation is a slicing of the system into the core system that comprises all functionality that interacts with the users and the adaptation subsystem that controls the choice of functionality [Fahrmair, 2005] (cf. Figure 5.3).

We claim that context adaptation is a special case of feature interactions. The core system slice is not faithful i.e. it depends on other inputs (those of the adaptation sub-system). Hence, we regard context adaptation as a feature interaction between the adaptation subsystem and the core system. While this may be intuitively clear we enforce this claim by a formal proof.

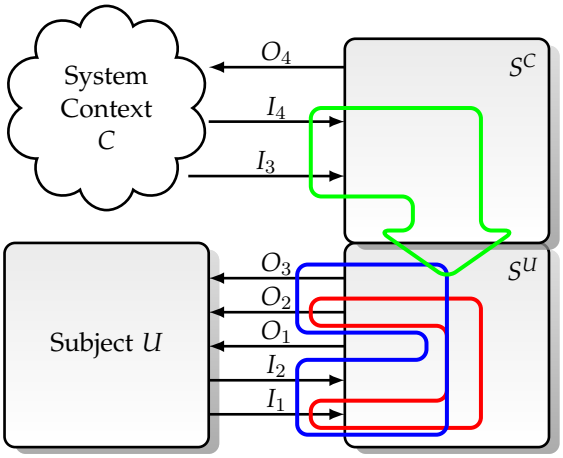


Figure 5.3.: context adaptation as a special case of feature interaction: the adaptation sub-system controls the behavior of the core system

---

THEOREM 1 (CONTEXT ADAPTATION IS A SPECIAL CASE OF FEATURE INTERACTIONS) *Any system with a structure that uses context adaptation with respect to a user-interface defines a not faithful slice which we call core system, that is in feature interaction with (is influenced by) the adaptation sub-system.*

PROOF 1.1 (OF THEOREM 1):
Let S be a system with interface $(I \rhd O)$ and $(I' \rhd O') \subset (I \rhd O)$ be the user interface. We proof the obligation by contradiction. Therefore, we assume that $S^U = S\dagger(I' \rhd O')$ is faithful. We start with the definition of context-adaptive systems.

$$(\exists\, x_1, x_2 \in dom(S) :$$
$$x_1 \lfloor_{I'} = x_2 \lfloor_{I'} \wedge \{y_1 \lfloor_{O'} \mid y_1 \in S.x_1\} \neq \{y_2 \lfloor_{O'} \mid y_2 \in S.x_2\})$$

With $x_1 \lfloor_{I'} = x_2 \lfloor_{I'}$:

$$\{y_1 \lfloor_{O'} \mid y_1 = S.x_1\} \subsetneq \{y_1 \lfloor_{O'} \mid y_1 \in S.x_1\} \cup \{y_2 \lfloor_{O'} \mid y_2 \in S.x_2\} \subseteq \{y \mid y \in S^U.x_1 \lfloor_{I'}\} \vee$$
$$\{y_2 \lfloor_{O'} \mid y_2 = S.x_2\} \subsetneq \{y_2 \lfloor_{O'} \mid y_2 \in S.x_2\} \cup \{y_1 \lfloor_{O'} \mid y_1 \in S.x_1\} \subseteq \{y \mid y \in S^U.x_2 \lfloor_{I'}\}$$

Hence,

$$\exists\, x \in dom(S) : \{y \lfloor_{O'} \mid y = S.x\} \neq \{y \mid y = S^U.x|_{I'}\}$$

which is a contradiction to the initial assumption that S^U is faithful. *q.e.d.*

We argue at the level of systems because we defined context-adaptive systems at that level in Chapter 3. Therefore, we consider the interface abstraction of a service specification. Hence, we reduce our considerations to systems S with an interface $(I \rhd O)$ and a slice $(I' \rhd O')$ rather than considering service specifications with interface $(I \rhd O, V)$ and a slice $(I' \rhd O', V')$. The result easily carries over to services as the following theorem shows.

THEOREM 2 (FAITHFULNESS OF SYSTEMS AND SERVICES)
For any $S = \Omega(F, \Lambda, \Upsilon)$

$$\neg \mathit{faithful}(S\dagger(I' \rhd O')) \Rightarrow \neg \mathit{faithful}(F\dagger(I' \rhd O', V')$$

PROOF 2.1 (OF THEOREM 2):
Let $S' = S\dagger(I' \rhd O')$ *and* $F' = F\dagger(I' \rhd O', V')$. *We start with the definition of a non-faithful slice at system level.*

$$\exists\, x_1 \in \mathit{dom}(S) : \{(y_{\lfloor O'}) \mid y \in S.x_1\} \neq \{y \mid y \in S'.x_{1\lfloor I'}\}$$

This is only possible if there exists a x_2 *such that*

$$(x_1 \neq x_2) \wedge (x' = x_{1\lfloor I'} = x_{2\lfloor I'}) \wedge (S.x_{1\lfloor O'} \neq S.x_{2\lfloor O'})$$

i.e., x_1 *and* x_2 *differ only at channels outside the slice and their respective outputs differ at least inside the slice. The following list contains the possible scenarios:*

- x_1 *and* x_2 *are identical: Obviously the outputs are identical and so are their projections*
- x_1 *and* x_2 *differ at channels within the slice: then* x_2 *does not contribute to the set* $\{y \mid y \in S'.x_{1\lfloor I'}\}$
- x_1 *and* x_2 *differ only at channels outside the slice. Then* $x_{1\lfloor I'} = x_{2\lfloor I'}$ *and the outputs according to inputs* x_2 *contribute to* $\{y \mid y \in S'.x_{1\lfloor I'}\}$. *However, joining the outputs only extends the set if* $S'.x_{1\lfloor I'}$ *and* $S'.x_{2\lfloor I'}$ *are different with respect to the slices outputs i.e. they are different from* $\{(y_{\lfloor O'}) \mid y \in S.x_1\}$

This implies that

$$\exists\, x_1, x_2 \in \mathit{dom}(S) : \{(y_{1\lfloor O'}) \mid y_1 \in S.x_1\} \neq \{(y_{2\lfloor O'}) \mid y_2 \in S.x_2\}$$

Furthermore, we know that $\exists\, t \in \mathbb{N}$ *such that* y_1 *and* y_2 *differ for the first time. We need this time to get from the infinite streams of a system to the finite streams of a service. Since* $S = \Omega(F, \Lambda, \Upsilon)$,

$$\exists (\sigma_1, o_1) \in \Lambda, \sigma_1', \sigma_2' \in \Sigma :$$
$$(\sigma_1, (x_1)_{\downarrow t}, (y_1)_{\downarrow t}, \sigma_1') \in [\![F]\!] \wedge (\sigma_1, (x_2)_{\downarrow t}, (y_2)_{\downarrow t}, \sigma_2') \in [\![F]\!]$$

and again,

$$(x_1 \neq x_2) \wedge (x' = x_{1\lfloor I'} = x_{2\lfloor I'}) \wedge (S.(\sigma_1, x_1)_{\lfloor O'} \neq S.(\sigma_2, x_2)_{\lfloor O'})$$

This demonstrates that we may not distinguish x_1 *and* x_2 *in their projections to* I' *but will get additional outputs in the set of outputs since* $((y_1)_{\downarrow t}, \sigma_1') \neq ((y_2)_{\downarrow t}, \sigma_2')$ *q.e.d.*

From Theorem 1 we conclude to treat context adaptation similar to feature interactions. For a service it is irrelevant if its behavior depends on a condition on the context or on another service. Both are dependencies on some additional inputs. Remember Example 3.3 in Section 3.1. We used the misuse protection as an example for a complex context and referred to this section to justify that a part of the input refers to an input of some (possibly different) user. The discussion above shows that this indeed is a reasonable choice.

It is possible to model most dependencies via mode transition systems but not yet always reasonable. As an example recall the adaptive steering examples in Section 3.2.2.1. With a continuous influence of the speed to the relation between the angle of the steering wheel and the actual steering angle, a large (possibly infinite) number of modes are necessary. This is not covered by MARLIN since the number of modes needs to be finite. In turn, a parametric adaptation that maps intervals of the speed to fixed relations between speed and steering angle fits well with mode transition systems in MARLIN.

A systems engineer has to find a good trade-off for modeling feature interactions. On the one hand, it is possible to make adaptations explicit via mode transition systems – which has the benefit of separating concerns and an enhanced analysis (cf. Section 6.2.2) – and using alternative composition to supplement behaviors.

5.2.2. Vertical relationships (hierarchic structures)

Multi-functional systems like context-adaptive systems serve multiple goals by offering multiple functions. A well-known approach in the requirements elicitation phase is defining goals and refining them by asking "what is necessary to achieve the goal". The goals thereby are stepwise refined into detailed requirements by adding information about providing the functionality [van Lamsweerde et al., 2001]. This builds up a hierarchy with functions serving the system goals on top and elaborating the actual functionality throughout the levels.

Similar to creating system architectures it is a matter of creative work and decent domain knowledge to refine the system goals into requirements. The nature of the composition operators of the formal specification language affects the organization of information during the refinement. Therefore, a formal language that deals with services as formalizations of requirements should support refining goals and requirements for its application domain by offering adequate composition operators for the services.

The service hierarchy in MARLIN supports three paradigms of organizing information by providing adequate composition operators: parallel, alternative, and conditional composition. They are motivated by the use of MARLIN for formalizing system specifications of context-adaptive systems. All paradigms serve the goal of focusing on separate aspects of the system behavior. Subsequently we point out the methodological use of the composition operators.

Parallel composition is useful for specifying different aspects of a functionality. This allows focusing on certain input channels and the effects at related output channels.

In the case study we describe the effect of the door triggers to the driver door, the passenger door, and the trunk door separately.

By parallel composition, services are effective at the same time. Services can control disjoint, overlapping, or congruent sets of channels and variables allowing to decompose complex logical functions into simpler usage functions[1] possibly affecting shared resources as well as decomposing usage functions into even smaller functions representing different views.

Alternative composition allows extending behaviors by adding new interaction patterns. Behaviors become either sequential or alternative. Alternative composition is a fine-grained mechanism to describe phases of execution.

Note that a behavior stops in states that do not offer adequate reactions to any inputs. The respective service is partial. Adding behaviors and reducing this partiality allows continuing interactions in this state at least for some inputs. This is why we say that behaviors sometimes become sequential if applying alternative composition.

Figure 5.4 illustrates the alternative composition of a behavior having virtually different effects. On the left hand side the composition extends the behavior allowing longer interactions. On the right hand side the composition adds just another alternative. Formally, both base on the same mechanism.

Conditional composition is coarser grained and separates conditions for offering a behavior and the actual description of the behavior. It combines the paradigms of parallel composition and alternative composition. Separating conditions and behaviors possibly includes separating the input interface. In addition, the behaviors describe phases of execution and therefore provide guarded alternatives.

Conditional composition aggregates and guards behaviors to control their activation. The mode-behaviors are independent and can be considered apart. Hierarchies of modes allow organizing complex condition guards. A transition at a higher level determines the possible modes and transitions at lower levels. This is useful to model dependent conditions.

To support the systems engineer with organizing the services and their relations we introduce the concept of a *service tree* similar to [Gruler and Meisinger, 2009] and [Broy, 2010]. We use the term "service-tree" to emphasize its preliminary status only relating services with sub-services without considering the used composition. The term *service-hierarchy* as defined later captures the actual relationships between services.

[1] We use the term logical functionality to describe a connected functional entity like ACC, active steering, radio, etc. In contrast we use the term usage function to describe a possible interaction like adjusting the volume, setting the speed, etc. Formally there is no difference

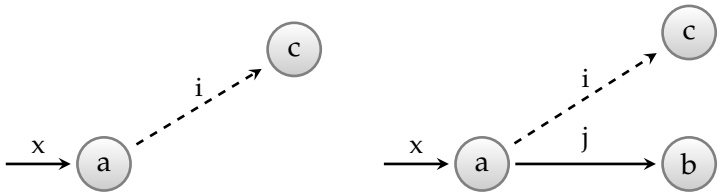


Figure 5.4.: Alternative composition of a transition from *a* to *c* with an action *j*. One extending the behavior "sequentially" and the other providing another alternative for the reaction

---

DEFINITION 36 (SERVICE TREE):
A service tree is a graph $G = (V, A, \varphi)$ *with nodes* $V \subseteq ID_S$ *(the set of service names) and edges* $A \subseteq V \times V$. A^* *is the transitive closure of A:* $(v_1, v_n) \in A^* \Leftrightarrow \exists\, v_2, \ldots, v_{n-1} : \forall\, i \in [1, n-1] : (v_i, v_{i+1}) \in A$. *We write*

$$path(v_i, v_j) = \{v_k \mid (v_i, v_k) \in A^* \wedge (v_k, v_j) \in A^*\}$$

for the set of nodes that occur on paths from v_i *to* v_j. *A graph that is a service tree additionally exhibits the following properties:*

- $\exists\, v \in V : \forall\, v' \in V : (v, v') \in A^*$ *(v is a root)*
- $\forall\, k, k' \in V : (k, k') \in A \Rightarrow \neg[(k', k) \in A^*]$ *(G is acyclic and directed, hence v is a unique root)*

$\varphi : V \rightarrow INTERFACE$ *is a function that associates an interface with each node of the tree such that*

$$\forall (k, k') \in A : \varphi(k) = (I \rhd O, V) \wedge \varphi(k') = (I' \rhd O', V')) \Rightarrow (I' \rhd O', V') \subseteq (I \rhd O, V)$$

For each $(k, k') \in A$ *the interface of the service associated with* k' *is a sub-interface of the service associated with k.*

We call the set of services k' *such that* $(k, k') \in A$ *the sub-services of k (subsequently denoted as set valued function sub(k)) and k a super-service of the* k' *(denoted as predicate super(*k'*)). Nodes k such that* $\nexists k' : (k, k') \in A$ *are leaves (denoted as leaf(k)) and relate to atomic services. The set of their sub-services is the empty set. Inner nodes relate to compound services.* □

A service tree holds no information about the actual nature of the composition nor the behavior of the services. It is simply a means to organize services. The service tree is an intermediate artifact to organize the structure of a compound service.

To allow a methodological treatment of (already known) dependencies we extend a service tree to a service tree enhanced with dependencies (similar to [Broy, 2010] and [Rittmann, 2008]) that relates those services that are in a (already known) relationship.

DEFINITION 37 (SERVICE TREE ENHANCED WITH DEPENDENCIES):
A service tree enhanced with dependencies (STeD) is a graph (G, ι). $G = (V, A, \varphi)$ is a service tree and ι is a relation between the nodes of the service tree such that:

$$\forall a, b \in V : (a, b) \in \iota \Rightarrow (a, b) \notin A^*$$

Dependencies may only exist for services in different branches. The relations in ι are directed arcs. In the rest of the thesis we refer to the service tree enhanced with dependencies if we speak of a service tree. □

The use of STeDs is similar to feature models for modeling product lines. A proper interpretation of the dependencies in ι allows using the STeD similar to FODA/FORM-Trees [Kang et al., 1990, 1998]. Rittmann describes the use of feature models for service based specifications [Rittmann, 2008] and Trapp uses FODA/FORM trees in [Trapp, 2005] to capture requires/excludes-dependencies between functions for context-adaptive systems. The STeDs are a bit more general because the dependencies can be anything that can be expressed in terms of valuations at channels. Deubler discusses some methodological relevant relations between services [Deubler, 2008].

STeDs still are just an intermediate artifact for building a system specification. The relation ι only captures the existence of dependencies without capturing their characteristic. The STeDs are the basis for deciding about resolving dependencies. The goal is to design a service hierarchy.

DEFINITION 38 (SERVICE HIERARCHY):
A service hierarchy is a tuple (G, Ψ) where G is a service tree and $\Psi : V \to \mathbb{F}$ is a mapping of nodes to service specifications such that either of the following conditions hold:

- $\forall (k, k_1), ..., (k, k_n) \in A : \Psi(k) = \Psi(k_1) \oplus ... \oplus \Psi(k_n)$ *(The behavior associated with k is an alternative composition of its children)*
- $\forall (k, k_1), ..., (k, k_n) \in A : \Psi(k) = \Psi(k_1) \otimes ... \otimes \Psi(k_n)$ *(The behavior associated with k is a parallel composition of its children)*
- $\forall (k, k_1), ..., (k, k_n) \in A :$
  $\Psi(k) = (\{m_1, \ldots, m_n\}, \delta, \Phi)$ *and* $\Phi : \{m_1, \ldots, m_n\} \to \mathbb{F};\ \forall i \in [1 \ldots n] : \Phi(m_i) = k_i$
  (k is a mode automaton with $k_1, \ldots, k_n$ as the behaviors associated with the modes)

In this hierarchy the nodes V of the underlying service tree are a super set of the nodes of the preceding STeD. The dependencies in the STeD map either to mode transition systems or to additional services in alternative compositions. □

The service hierarchy models the service/sub-service relation together with resolved mutual dependencies of the services and resembles the hierarchic structure of a MERLIN specification with parenthesis. All yet known dependencies are resolved by applying an appropriate measure like alternative or conditional composition.

The structure of a service hierarchy is not unique. Besides the characteristics of the composition operators the application domain affects the composition of a functional specification. There are many possibilities to compose services into more complex ones. Regard the case study in appendix B.1. One may define a compound service *TrunkOperation* out of services F_2,F_3,F_4 and F_5 as well as a compound service *CarAccess* from services F_2, F_4, F_6, F_7, and F_8. The good news is that the algebraic laws from Section A.3 allow restructuring specifications and relating them. However, an educated choice could result in a more or less well-arranged hierarchy and thus supports the validation.

5.3. Preparing contexts

Mode transition systems are the core concept for specifying context-adaptive systems in MARLIN and include a number of methodological considerations. In this section we discuss aspects of contexts having an impact on mode transition systems and their handling in the contextual requirements chunks.

Contextual requirements chunks are the starting point for the modeling of context-adaptive systems and carry already important information about contexts that affects modeling in MARLIN. Other information possibly must be supplemented to the contextual requirements chunks after a first iteration of modeling and analysis like possible feature interactions that are initially unknown. The following discussions help during transferring the contextual requirements chunks to mode transition systems. We start with relations of services and their contexts and continue to prepare the contexts considering the dependencies. This is to prepare an approach that we call unbundling (cf. Section 5.4) to systematically transfer contextual requirements chunks along the lines of their contexts into mode transition systems.

5.3.1. Capturing feature interactions

In Section 5.2.2 we discussed feature interactions from a theoretical point of view and related feature interactions to partiality and non-determinism (depending on the point of view). In practice, feature interactions appear often implicitly. Services formalize use-cases that describe an isolated aspect of the intended system – often independently from other viewpoints of stakeholders. Their integration requires considering possible interactions and interferences.

We model known feature interactions (already clear from the viewpoints or becoming known after analysis) either by alternative composition of a service that represents the feature interactions or by mode transition systems. Mode transition systems fit well

if the nature of the feature interaction allows a discrete separation of behavioral patterns.

5.3.1.1. Relating feature interactions to slices of the influencing service

The parallel composition allows using internal communication to pass information from an influencing service to an influenced one. Such inter service relations are, e.g., described in [Zave and Jackson, 2000; Pulvermueller et al., 2002] and [Deubler, 2008]. However, in MARLIN we relate any feature interaction to additional system inputs.

Subsequently, we investigate settings, where a STeD describes services that influence other services and the nature of the influence is discretely changing the behaviors of the influenced service. Broy calls this influence mode communication [Broy, 2010]. We show a reorganization of the services and their related contexts to create a specification without inter service communication.

We assume that dependencies on context information are already captured in the contextual requirements chunks and that the contexts are arranged accordingly. Without restrictions, for the upcoming considerations we assume a service A influencing a service B with $B_1, B_2, \ldots B_n$ as variants of the service behavior of B.

The goal of this section is preparing the unbundling of the contextual requirements chunks. The result of the unbundling will be an auxiliary service B_{AUX} that is a mode transition system equipped with modes $M_1, M_2, \ldots, M_n$ that hold the sub-behaviors of B. The preparation properly manipulates the contexts of the respective services B_1 to B_n. The actual creation of the mode transition system describe Sections 5.4.1 and 5.4.2.

If A influences B to change its behavior, there is an information flow from A to B (see Figures 5.5a). We distinguish two kinds of feature interactions from A to B:

1. feature interactions that directly relate to inputs of A (i.e., they are a direct result of excluding the these inputs from the slice) and the nature of the influence easily can be reduced to these inputs. Think of an incoming call that mutes the radio in example 5.2.

2. feature interactions which refer to outputs of A and the nature of the feature interaction cannot be related easily to inputs of A or the relation involves a complex calculation. Think of a navigation system that occasionally mutes the ratio to announce a direction.

In any case, we need to know the nature of the feature interaction. The elicitation of this knowledge usually requires manual analysis and domain knowledge. We consider the need for this analysis as an advantage. Feature interactions can cause unintended behavior or contradictions. Analyzing the feature interactions enhances the system understanding and finally raises the quality of the system.

5.3.1.2. Feature interaction attributed to additional inputs only

We aim at introducing an auxiliary service B_{AUX} as a mode transition system that we compose in parallel to *every* occurrence of A. This is necessary because mode transi-

tion systems maintain their history variable only while they are active. If we miss to compose it to all occurrences of A and create a situation where A is active but B_{AUX} is inactive, B_{AUX} will possibly decide on outdated data in later transitions. If B shall be inactive, we use a mode in B_{AUX} with $SAME_{V_B}$ as a mode-behavior.

Figure 5.5 illustrates the situation. On the left side, Figure 5.5a illustrates the starting situation with A influencing B. Only some inputs of A affect B without complex calculation over these channels.

After an investigation, we identify channels $I_{FI} = \{i_j, \ldots, i_k\}$ to cause the influence of A to B (i_2 and i_3 in the figure). In this setting calculations of A are unimportant for B. Only some inputs of A are missing in the slice of B. As a result B is not faithful in $A \otimes B$.

Service B shall be transferred into a mode transition system with modes $M_1, M_2, \ldots, M_n$ (M_1 and M_2 in the illustration in Figure 5.5b) and assigned mode-behaviors $B_1, B_2, \ldots, B_n$ (B_1 and B_2 in the figure). The modes change according to conditions $c_1, c_2, \ldots c_m$ that represent the conditions on the input channels I_{FI}.

To prepare the creation of this mode transition system we extend the contextual requirements chunks $(Name, (IN, OUT), Behavior, Scenario)$ with a table that captures the different variants of B by introducing entries $B_1, \ldots, B_n$:

$(Name_1, (in_1, out_1), B_1, SC_1)$
$\ldots$
$(Name_n, (in_n, out_n), B_n, SC_n)$

The entry and exit conditions must match the conditions of the feature interaction that activate the variants of B. Since we assume that the B_i are alternative variants it is reasonable to require:

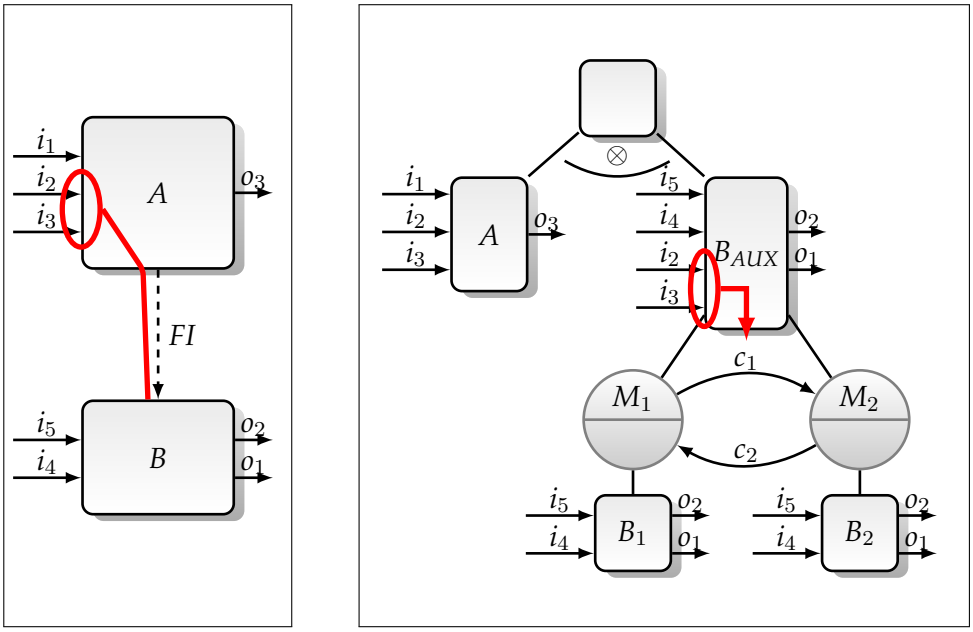
$$in_i \Rightarrow \bigwedge_{j \in \{1,\ldots,n\} \setminus \{i\}} out_j$$

which easily can be achieved by choosing the exit conditions as:

$$out_i \stackrel{def}{=} \bigvee_{j \in \{1,\ldots,n\} \setminus \{i\}} in_j$$

This condition ensures that at most one of the variants is active at the same time.

This approach reveals the dependencies of B and makes them explicit. It decouples the influencing service and the nature of the influence while maintaining the descriptions of the original sub-behaviors of B. As a result, we can exchange the formerly influencing service A or alter it without any risk of affecting the new service B_{AUX}. Even more important, the services $B_1, B_2 \ldots B_n$ exist separately without taking any aspect of A into account. By separation of concerns and a modular consideration of services this approach contributes to the understanding of correlations especially if multiple services influence the behavior of B.



(a) A influencing B **(b)** FI modeled as modes

Figure 5.5.: Modeling Feature Interaction from A to B as modes

In addition to the feature interaction a dependence on some context may exist for either of the services. In that case, a nesting of mode transition systems helps. The handling of nested conditions already in the contextual requirements chunks explains Section 5.3.2.

5.3.1.3. Complex feature interactions without clear relation to inputs

Transition guards of mode transition systems are restricted to regular languages. Any feature interaction that has a higher complexity cannot be expressed directly as a condition at a mode transition. If the nature of the feature interaction remains complex even after an in-depth analysis and the direct mapping to contexts is awkward, mode transition systems allow another solution to model complex feature interactions (to the cost of being less accessible for analysis).

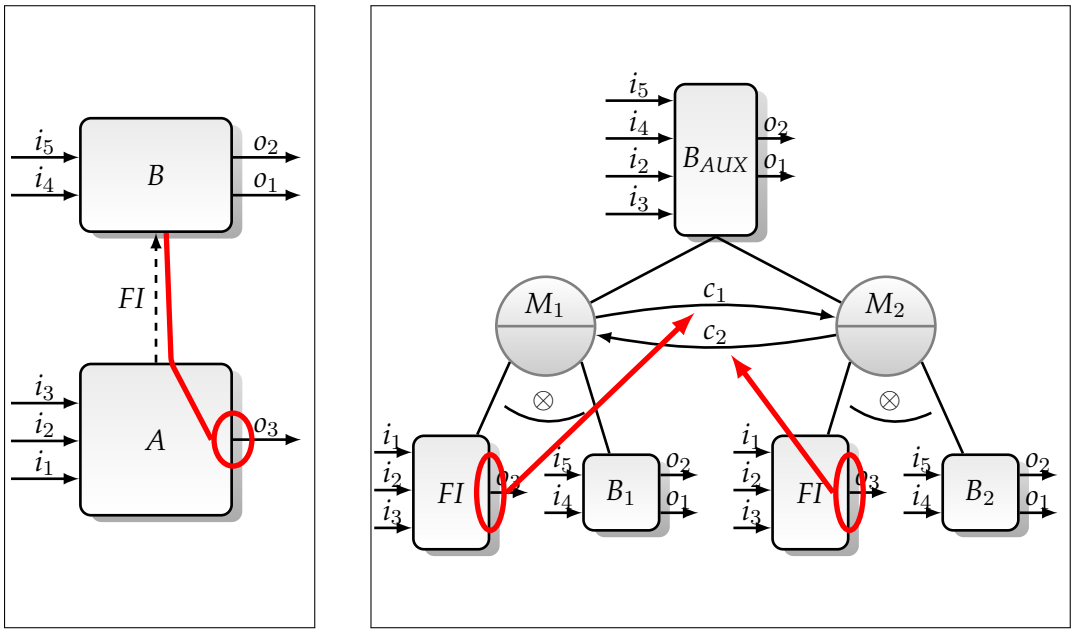
Usually, the essence of the interaction corresponds to a slice of A with respect to those channels that carry the information that influences B. Let us call the channels I_{FI} with $I_{FI} \subseteq I_A$ and O_{FI} with $O_{FI} \subseteq O_A$. The slice that is responsible for the feature interaction is $FI = \dagger(I_{FI} \triangleright O_{FI}, V_{FI})$ with a properly chosen set of variables $V_{FI} \subseteq V_A$.

We assume that the analysis reveals this slice i.e. we know the additional information that controls the behavior of B. Therefore, we are able to write conditions $(c_i)_{IN}$ and $(c_i)_{OUT}$ over the channels of the slice FI that model the influence on B. The trivial slice that carries this information is A itself. However, usually the essence of a feature interaction is much simpler and the conditions are as well.

Mode transition systems allow taking outputs of their own mode-behaviors into account: transition guards of a mode transition system are predicates over channels $I \cup O_{\Phi(m)}$. This allows integrating FI into the auxiliary service B_{AUX} as a part of the mode-behaviors. The transition guards refer to the outputs of FI. Formally, we replace the variants B_i of B by services $B_i' = B_i \otimes FI$.

This approach is similar to distributing parallel composition over mode transition systems as presented in Appendix A.3.2. FI moves to the mode-behaviors of the mode transition system.

This operation is without consequences for the behavior FI and all B_i. Only the contexts are affected. Figure 5.6 illustrates the approach for services A with $FI = A\dagger(I_{FI} \rhd O_{FI}, V_{FI})$ and B with two variants of B.



(a) A influencing B

(b) A as an inherent part of the mode transition system

Figure 5.6.: Integration of A into the mode transition system of B for the sake of modeling complex feature interaction

Similar to the approach for feature interactions attributed to inputs we create a contextual requirements chunk table for the variants of B. The entry and exit conditions refer to the outputs of FI. Note that it may be necessary to define some extra output channel of FI that is not part of the original service A as, e.g., proposed in [Broy, 2010]. This modifies FI such that it is no longer a real slice of A. Furthermore note that such an extra output allows restricting the contexts to simple contexts: any complex sequence of messages can already be handled within FI to send only a single signal encoding the desired variant of B.

Let $(NAME_{FI}, (in_{FI}, out_{FI}), FI, -)$ be the essence of the feature interaction. We change

the contextual requirements chunks that describes the variants of B such that:

$$(Name_{B_i}, (in_i, out_i), B_1, SC_i) \dashrightarrow$$
$$(Name_{B_i'}, (in_i \wedge c_i(O_{FI}) \wedge in_{FI}), (out_i \vee \neg c_i(O_{FI}) \vee out_{FI}), B_1 \otimes FI, SC_1)$$

We combine the contexts of FI with the variants of B to allow a proper treatment of feature interactions in settings where A and B have different contexts. If A is inactive, there cannot be any feature interaction from A to B. As a result there shall be no influence from FI to the mode transition system.

If B can be active without A using the context of A as the context of FI and combining it with the context of the variants of B ensures that no variant of B is active if A isn't either. Unbundling (see Sections 5.4.1 and 5.4.2) maps undefined behavior to this setting and the analysis in Section 6.2 reveals these situations to allow addressing them properly.

5.3.2. Hierarchic mode transition systems

If contexts are not overlapping but one includes another we can use a hierarchy of mode transition systems. Sometimes it is already clear from the contextual requirements chunks that some functions or sets of functions and their switching is only available in certain contexts.

The trunk opening by gesture depending on the availability of a key is an example that is only available if the battery has enough capacity. We may want to model this as a hierarchy. At an upper level we describe the switching depending on the battery state. This upper level either activates the gesture handling of the trunk or completely deactivates it. At a lower level we model the dependency on the key.

We shortly summarize observations for the hierarchic modeling of contexts. Formally identifying such dependencies in sequences of messages is a broad topic and out of scope of this thesis. In general we need to find contexts

$$(in_{A_1} \Rightarrow in_B) \wedge (out_B \Rightarrow out_A)$$
$$\dots$$
$$(in_{A_n} \Rightarrow in_B) \wedge (out_B \Rightarrow out_{A_n})$$

to identify settings, where all A_i are services that are only available in the context of B.

For simple contexts the situation is straight forward. Simple contexts are formulas in propositional logic over the current inputs. The function is active as long as the context condition is satisfied. The contexts of the candidates may be transferred into conjunctive normal form (CNF). It is possible to factor out common terms of the context conditions. These terms are gathered for modeling them in an upper level in a hierarchy. Figure 5.7 illustrates the approach.

In the figure, two contexts A and B share a common condition which is factored out as C. In the illustration, C is larger than A and B together, indicating that there is another context within C that satisfies neither the condition of A nor that of B (in the figure we

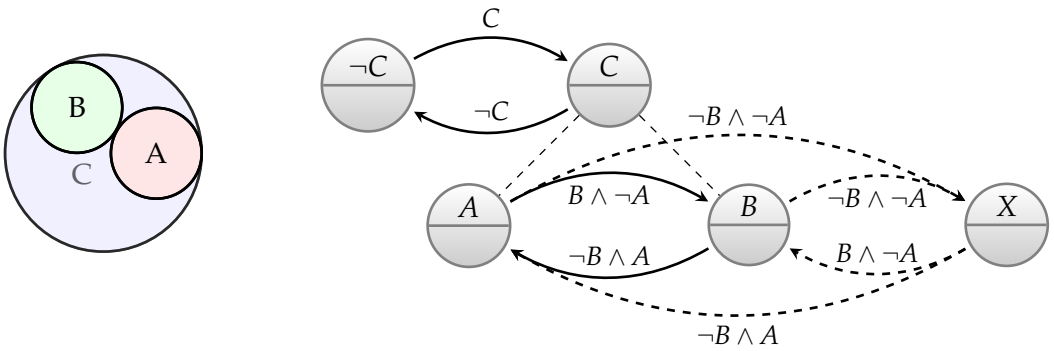


Figure 5.7.: Modeling a simple hierarchy of contexts in a hierarchic mode transition system

use X to represent this setting). In a hierarchic mode transition system the enclosing context of C appears at an upper level. We use a second mode transition system as a mode-behavior at a lower level, to capture the switching between A, B and X.

The situation for complex contexts is more complex. In general identifying shared aspects of contexts that allow a modeling as a hierarchy involve identifying common prefixes and/or subsets of streams characterized by the entry and exit conditions. For our considerations we assume a simpler and probably more usual setting: the contextual requirements chunks are already grouped and annotated with general conditions for sets of functions. We apply unbundling to the groups before applying unbundling to the elements of each group separately.

	Name	Scope		Requirement
		IN	OUT	
Common scope		c1	c2	
m_1	1	in1	out1	function 1 (f_1)
	2	in2	out2	function 2 (f_2)
Common scope		c3	c4	
m_2	3	in3	out3	function 3 (f_3)
	5	in4	out4	function 4 (f_4)

Figure 5.8.: Clipping of contextual requirements chunks with contexts grouping sets of functions

Figure 5.8 shows an example for a possible contextual requirements chunks specification with two sets of functions. A common context groups the sets. Figure 5.9 shows a possible mapping for the contextual requirements chunks in Figure 5.8 after applying unbundling first to the group contexts and in a second step to the elements of the groups separately. We assume that $c1 \Leftrightarrow c4$, $c3 \Leftrightarrow c2$, $in1 \Leftrightarrow out2$, $in2 \Leftrightarrow out1$, $in3 \Leftrightarrow out4$, $in4 \Leftrightarrow out3$ i.e. the exit conditions of each of the contexts correspond to the

entry conditions of their counterparts. Otherwise we get additional modes.

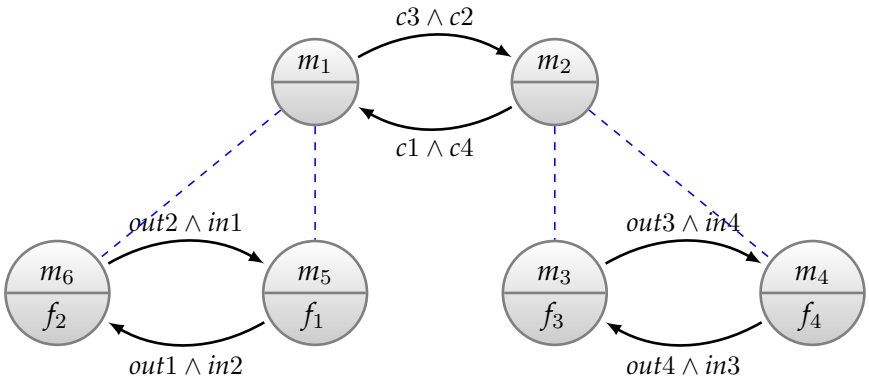


Figure 5.9.: Hierarchic mode transition system mapping the contextual requirements chunks of Figure 5.8

In the context of hierarchic mode transition systems the naming of mode transition system variables is important. In general – as with any variable – it is possible to use the same mode transition system variables in two different mode-behaviors that can be mode transition systems as well.

Using shared mts-variables indicates an *identical nature* of the sensitivity to contextual aspects of two mode transition systems. A simple sensitivity to the same contextual inputs is no sufficient indicator for the use of shared mts-variables. For example, the automatic door closing of the case study (Requirement 1) depends on the speed. The active steering in Example 3.7 depends on the speed as well. However, we hardly intend the doors to change the behavior together with the steering.

EXAMPLE 5.6 (USE OF SHARED MTS-VARIABLES)
We regard three mode transition systems A, B_1 and B_2 as illustrated in Figure 5.10. The mode transition system A has modes m_1 and m_2 and mode-behaviors B_1 and B_2 that are mode transition systems as well. By intention, we choose the same set of modes (n_1, n_2 and n_3) for both subordinate mode transition systems B_1 and B_2 and use the same mts-variable PC_B (in the figure this is just indicated by the shared use of the mode names). In this scenario the mode switches that are possible between modes n_1, n_2 and n_3 depend on the mode of A.

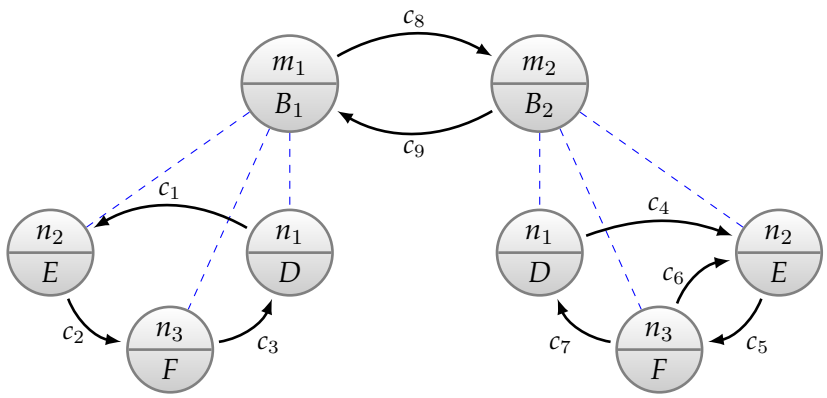


Figure 5.10.: Hierarchic mode transition systems with shared mts-variables used by the two sub-ordinate mode transition systems

The decision about sharing mode transition system variables between separate mode transition systems is an explicit design decision. Neither the requirements in the contextual requirements chunks nor their contexts carry information that allow a systematic transformation into corresponding mode transition systems. For any two mode transition systems we assume that they share either both, the mode pointer *and* the history variable or none of them.

5.3.3. Processes and work flows

Another aspect of contexts needs our attention. (Business) processes or work flows describe sequences of activities. As a particular property, the activities have a well-defined order in addition to conditions of finishing or starting activities. In workflows multiple sequences of allowed activities exist. Choosing the appropriate successor depends on context information and outcomes of the activities. The activities are supported by different behaviors of the system, e.g., to support users with their tasks. Examples are a mars lander running through different phases of the landing sequence or a surgery with a well defined activity structure [Leuxner et al., 2009b]. We pick up the surgery in Example 5.7.

Activities and their order can be mapped to modes. For this purpose, we make a direct reference to other modes in contexts in addition to conditions on input- and output channels. These references are predecessor and successor modes.

EXAMPLE 5.7 (WORK FLOW)
The case study of this thesis contains no reasonable sequence of activities. Therefore, we refer to an example that is originally presented in [Leuxner et al., 2009a] and [Leuxner et al., 2009b]. The example describes a surgery by sequential activities. The system observes seven contextual aspects in a work flow of 270 elements. The nature of the observed context and the number of activities result in many combinations that are ambiguous with respect to the applicable context i.e. it is impossible to identify the current phase of the surgery only from the observed information.

Information about the history of activities complements the contextual information to disambiguate the decisions. The information about the history of activities is an additional context information. Figure 5.11 illustrates a small clipping of the surgery's work flow. In the clipping a

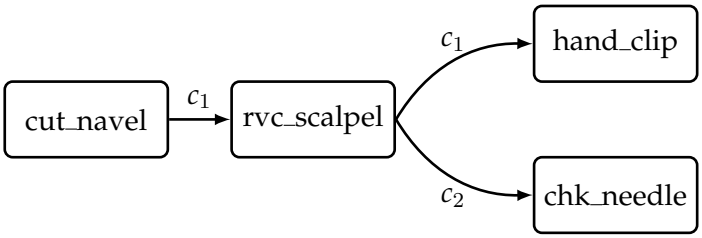


Figure 5.11.: A clipping of a work flow description of a surgery setting, describing some activities for opening the body

condition c_1 (certain combination of observed sensor information and events) leads from activity `cut_navel` *(the surgeon cutting the navel of the patient) to activity* `rvc_scalpel` *(the nurse receives the scalpel). The same sensor observations can be made after activity* `rvc_scalpel` *which leads to activity* `hand_clip` *(the nurse hands over a clip). The condition c_1 is insufficient to determine an activity. The predecessor (i.e., the knowledge about already completed activities) provides additional information to determine the next activity.* ♣

The example shows that work flows resemble mode transition systems. The system supports the phases with different functions. Selected information indicates the end of a phase and indicates the follow-up phase. Therefore, we model processes or work flows as mode transition systems. The transitions between the activities become mode transitions using dedicated information at the transition guards to indicate the end of a phase and to select the next one. Each mode capsules the behavior that is characteristic for a phase.

Sometimes the mapping of workflows to mode transition systems is straight forward – especially if the workflow has already been modeled. However, to underline the similarities between work flows and mode transition systems we demonstrate the transformation of work flows into mode transition systems by unbundling. The approach creates a mode transition system that considers the right order of modes as well as the conditions for (de)activation.

In this setting we augment the contextual information in the CRC by process or work flow information. Remember from the example that formally the information

about a work flow is additional context information. We model the dependence on this additional context information by additional aspects *pre* and *post* that become part of the entry and exit conditions of a requirement's context.

The elements are set-valued and store all possible predecessors/successors. Therefore, we assign names from a set $NAMES_{contexts}$ to all existing contexts and assign $\mathcal{P}(NAMES_{contexts})$ as type to *pre* and *post*. For later use we introduce the functions $pre : NAME_{contexts} \rightarrow \mathcal{P}(NAMES_{contexts})$ and $post : NAME_{contexts} \rightarrow \mathcal{P}(NAMES_{contexts})$ that return the set of *pre*/*post*-elements for a requirement.

By neatly handling combinations of *pre*- and *post*-elements together with other conditions we can model any complex relation between work flows and conventional transition conditions. We will consider this in the upcoming description of unbundling. For more detailed information on workflow modeling and a sophisticated formalism hat involves an easy description of parallel execution of tasks we refer to [Leuxner et al., 2009b, 2010].

EXAMPLE 5.8 (WORK FLOW (CONT.))
The contexts of the CRCs with respect to Example 5.7 read according to Table 5.12 after the extension:

Name (Number)	Context		Requirement
	IN	OUT	
cut_navel	TRUE	$c_1 \wedge post = \{rvc\_scalpel\}$	not yet defined
rvc_scalpel	$c_1 \wedge pre = \{cut\_navel\}$	$(c_1 \wedge post = \{hand\_clip\}) \vee (c_2 \wedge post = \{chk\_needle\}$	not yet defined
hand_clip	$c_1 \wedge pre = \{rvc\_scalpel\}$	not yet defined	not yet defined
chk_needle	$c_2 \wedge pre = \{rvc\_scalpel\}$	not yet defined	not yet defined

Figure 5.12.: Clipping of the contextual requirements chunks of a surgery case study

5.4. Systematic construction of mode transition systems

The requirements elicitation for context-adaptive systems results in a collection of requirements and respective contexts. Usually, a number of requirements in combination describe a single logical function, possibly including different contexts this function has to adapt to. Other requirements are only loosely related because they belong to different logical functions. It is a natural choice to integrate the requirements for a single

logical function into a (compound) service – possibly a mode transition system – while composing different logical functions in parallel.

In the contextual requirements chunks the contexts are yet unrelated. Sometimes the contexts actually overlap, indicating that the requirements occasionally must be satisfied at the same time. Therefore, the contexts do not directly reflect their usage in a mode transition system. Creating mode transition systems that handle overlapping contexts requires the disentangling of overlapping of contexts. We call this *unbundling*.

Recall that a CRC specification is a list

$$(s_1,\ (in_1, out_1),\ bh_1,\ sc_1)$$
$$\ldots$$
$$(s_m,\ (in_m, out_m),\ bh_m,\ sc_m)$$

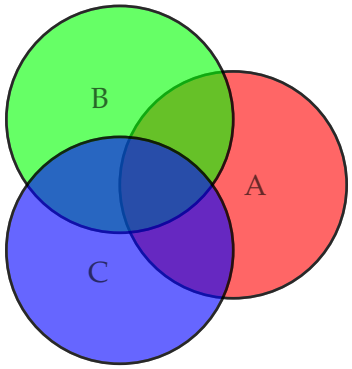
For unbundling, we assume that the steps of preparing the contexts as presented in Section 5.3 are finished and the contextual requirements chunks are modified accordingly.

5.4.1. Simple unbundling

By introducing simple contexts in Section 3.1.2, we provided a simplified means to define a certain kind of contexts. Similar, it is possible to provide two methods of unbundling: a simpler one that is appropriate for simple contexts only and a more complex one that covers all kinds of contexts. We start by explaining the simple unbundling.

Creating disjoint equivalence classes Simple contexts correspond to propositions in propositional logic and define equivalence classes of valuations of current inputs at considered channels. A simple context can be represented as the Cartesian product of information at the relevant input channels and – if applicable – the *pre/post* element as an additional condition. As long as the considered current inputs satisfy the condition the assigned function is effective. Otherwise it is ineffective. Since the *pre/post* elements are no system input we treat them in a different way that guards the construction of the mode transition system.

All usual set-relations between contexts exist. We define unbundling of simple contexts with respect to these relations. Initially, the contexts in the contextual requirements chunks relate to some services. Let $NAME_{contexts} = \{s_1, \ldots, s_m\} \subseteq NAMES$ be the names of the considered contexts. These names shall correspond to the names in the contextual requirements chunks.



$pre(A) = \{B\}$
$pre(B) = \{C\}$
$pre(C) = \{A\}$

Figure 5.13.: A representation of three contexts with overlapping areas and the corresponding *pre*-elements

---

EXAMPLE 5.9 (OVERLAPPING CONTEXTS)
We illustrate simple unbundling with a simple example. Initially we assume three overlapping contexts A, B and C and represent them in a two-dimensional way.

In the case study (cf. Section B.1) the contexts of requirements R02 (Trunk is opened on gesture) and R07 (all doors unlock if driver door is touched) overlap. Requirement R02 has a context

$$S2 \stackrel{def}{=} (\texttt{V} = \texttt{lo} \wedge \texttt{UBat} = \texttt{hi} \wedge \texttt{KPos} = \texttt{tr})$$

Requirement R07 has a context

$$S7 \stackrel{def}{=} (\texttt{V} = \texttt{lo} \wedge \texttt{KPos} = \overline{\texttt{away}}).$$ ♣

Without restriction A and B are two arbitrary contexts. Step one of unbundling applies the following cases to any two pairs of contexts recursively until reaching the base case. We define an auxiliary function

$$SRVCS : NAMES_{CONTEXT} \rightarrow \mathcal{P}(NAMES_{SERVICES})$$

that assigns the services that are active to a named context and manipulate it during unbundling. For all $(n, (in, out), bh, sc)$ initially $SRVCS(s_n) = f_n$.

Disjointness This is the base case of the recursion. Two contexts A and B have no intersection and hence the associated services never are active at the same time. We write $A \parallel B$. Formally $A \parallel B \Leftrightarrow A \cap B = \varnothing$. In such a case no (further) unbundling is necessary. If any two pairs of contexts are in this relation, the unbundling is finished.

Overlapping Two contexts A and B share some elements. Hence, the associated services can be active at the same time. We write $A \between B$. Formally $A \between B \Leftrightarrow A \cap B \neq \varnothing$.

In such case a transformation is necessary. We split the overlapping contexts A and B into three contexts $A' = A \backslash B$, $B' = B \backslash A$ and a new one $C = A \cap B$. We assign all services of A to A': $SRVCS(A') \mapsto (SRVCS(A))$; and those of B to B': $SRVCS(B') \mapsto (SRVCS(B))$. We assign the union of the sets of services of A and B to C: $SRVCS(C) \mapsto (SRVCS(A) \cup SRVCS(B))$. In addition we update the *pre*- and *post*-elements. In all contexts with *pre*/*post*-elements containing A or B we replace the occurrences by elements A', C and B', C respectively. The *pre*/*post*-elements of the new context C becomes $pre \stackrel{def}{=} \{A', B'\} \cup pre(A') \cup pre(B')$ and $post \stackrel{def}{=} \{A', B'\} \cup post(A') \cup post(B')$. In set $NAMES_{contexts}$ we change A to A', B to B', and add C.

Congruence The contexts A and B for two requirements are congruent. Formally $A = B$. In such a case we join the equivalent contexts into a new context C and join the services of A and B: $SRVCS(C) \mapsto (SRVCS(A) \cup SRVCS(B))$. Again we update the *pre*- and *post*-element by replacing each occurrence of A or B with the new context C. In the set $NAMES_{contexts}$ we replace A and B by C.

Subset The subset relation is a special case of overlapping and is treated accordingly.

After some iterations, all remaining contexts become disjoint. As an important aspect we include a special context to the consideration: we call it $\perp$ and assign $\{\}$ to it. $\perp$ represents the complete context space: $context_{in}(\perp) = TRUE$ and $context_{out}(\perp) = FASLE$. This guarantees the consideration of all possible contexts including areas that are not covered by any other functions' context. $\perp$ is always active. $\{\}$ is neutral with respect to union. Therefore, this operation has no effect on the other existing behaviors.

EXAMPLE 5.10 (OVERLAPPING CONTEXTS CONT.)
After applying unbundling the overlapping areas become separate partitions with new names. For the sake of this example we omit $\perp$ and assume that the original A, B and C already cover all possible contexts.

We map the considerations to the requirements R02 and R07 of the case study. We use and introduce artificial contexts

$$S02' \stackrel{def}{=} \neg S02 \text{ and } S07' \stackrel{def}{=} \neg S07$$

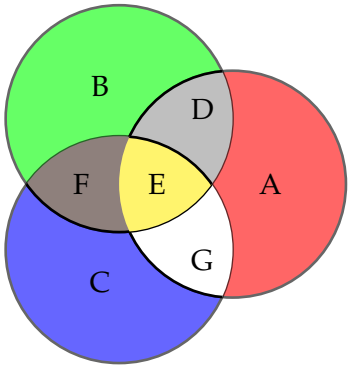
that allow us to show the effects of unbundling. The unbundling results in three contexts which we call A, B and C with

$$A = S02 \wedge S07, \qquad B = S02' \wedge S07, \qquad C = S02' \wedge S07'$$

Note that $S2 \wedge S07'$ is impossible because S02 is a subset of S07. The assigned services are:

$$A : \{F_{02}, F_{07}\}, \qquad B : \{F_{02'}, F_{07}\}, \qquad C : \{F_{02'}, F_{07'}\}$$

where $F_{02'}$ and $F_{07'}$ are artificial functions that define the behavior if S02 and S07 are not effective. ♣



$$
\begin{aligned}
A &: pre = \{B, D, E, F\} \\
B &: pre = \{C, F, E, G\} \\
C &: pre = \{A, D, E, G\} \\
D &: pre = \{B, C, D, E, F, G\} \\
E &: pre = \{A, B, C, D, E, F, G\} \\
F &: pre = \{A, C, D, E, F, G\} \\
G &: pre = \{A, B, D, E, F, G\}
\end{aligned}
$$

Figure 5.14.: The areas are partitioned such that each new area is non-overlapping with any other area

Creating a preliminary mode transition system In the second step, we establish the mode transition system $(\mathbb{M}, \delta, \Phi$ by creating the modes according to the different partitions in the set $NAMES_{contexts}$. For every context $s \in NAMES_{contexts}$ we create a mode m_s.

$$\mathbb{M} \stackrel{def}{=} \{m_s \mid \exists s \in NAMES_{contexts}\}$$

For a context s (and the corresponding mode m_s) with assigned service f_s we define the service-mapping function as.

$$\Phi \stackrel{def}{=} \{(m_s, f_s) \mid s \in NAMES_{contexts}\}$$

Finally we define the transition relation. Initially, the transition relation relates all pairs of modes. The transition labels describe the change from one context to the other one.

$$\delta \stackrel{def}{=} \{(m_{s_i}, (context_{in}(s_j), \epsilon), m_{s_j}) \mid m_{s_i}, m_{s_j} \in \mathbb{M}\}$$

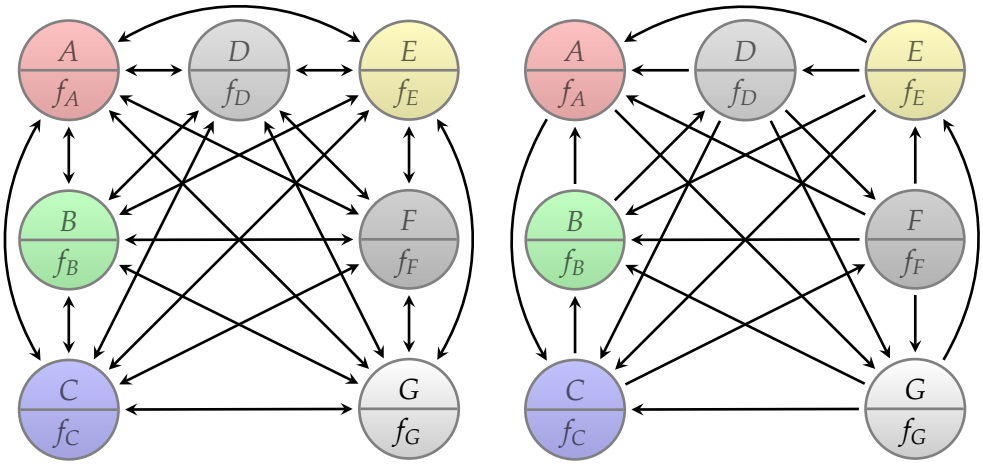
We can safely use the reset flag in the transition because all transitions deal with simple contexts.

Pre and post If present, we consider the *pre*- and *post*-elements. We remove all incoming transitions of a mode except those that start in another mode that is contained in the *pre* element of the mode under consideration. We handle the *post*-elements in a similar way.

Formally we keep only transitions

$$
\begin{aligned}
&(m_{s_i}, (context_{in}(s_j), \epsilon), m_{s_j}) \in \delta \Leftrightarrow \\
&\quad m_{s_i} \in pre(m_{s_j}) \wedge m_{s_j} \in post(m_{s_i})
\end{aligned}
$$

EXAMPLE 5.11 (OVERLAPPING CONTEXTS CONT.)
Figure 5.15 omits the labels of the mode transition system transitions. Figure 5.15a illustrates the transition into the mode transition system without considering the pre-elements. Figure 5.15b present the situation after considering the pre-elements.



(a) The contexts integrated into a mode transition system

(b) The contexts being integrated into a mode transition system, now considering *pre*

Figure 5.15.: Transfer into a mode transition system after unbundling

For the clipping of the case study creating the mode transition system MTS is straight forward.

$$MTS \stackrel{def}{=} (\{M_A, M_B, M_C\}, \delta, \Phi) \qquad with$$

$$\Phi \stackrel{def}{=} \{(M_A, \{F_{02}, F_{07}\}), (M_B, \{F_{02'}, F_{07}\}), (M_C, \{F_{02'}, F_{07'}\})$$

$$\delta \stackrel{def}{=} \{(M_A, (B, \epsilon), M_B), (M_A, (C, \epsilon), M_C), (M_B, (A, \epsilon), M_A),$$
$$(M_B, (C, \epsilon), M_C), (M_C, (B, \epsilon), M_B), (M_C, (A, \epsilon), M_A)\}$$

The case study describes no work flow and no pre/post-elements appear. A possible natural order of contexts that prevents the direct transition between modes may exist, e.g., for different levels of speed. In that case, a direct transition from low-speed to high-speed without visiting mid-speed is impossible. ♣

5.4.2. General unbundling

The unbundling for contexts with complex entry and exit conditions is a generalization of the unbundling for simple contexts. It is an adaptation of the product automaton construction to the peculiarities of mode transition systems. The general procedure includes three steps: I) construction of basic mode transition systems, II) integration of mode transition systems, and III) handling *pre/post* conditions.

Construction of basic context-automata As a first step, for each requirement (without restriction we call it n) we create a basic mode transition system. Such a basic mode transition system has exactly two modes. One mode carries the service that formalizes the requirement i.e. the respective service is active. The other mode represents yet undefined behavior if the context is ineffective. We refer to these modes with the name of the requirement m_n and $m_\perp$ respectively.

We call the mode relating to the service that formalizes the requirement the active-mode. The transition from $m_\perp$ to the active-mode is labelled with the context's entry condition. The transition from the active-mode to $m_\perp$ is labelled with the exit condition. Because this basic mode transition system formalizes the activation of a single context we call it a context-automaton.

Formally, let $(n, (in, out), bh, sc)$ be a requirement from the contextual requirements chunks and f_n be the service that formalizes the requirement bh. We create an automaton (M, δ, Φ):

- $M = \{m_\perp, m_n\}$
- $\delta = \{(m_\perp, (context_{IN}(n), \rho), m_n), (m_n, (context_{OUT}(n), \rho), m_\perp)\}$
- $\Phi = \{(m_\perp, \{\}), (m_n, \{f_n\})\}$

The fact that $\{\}$ is neutral in a set union operation is important for the integration of context-automata. We use ρ to continue the history variable. Using ϵ requires additional considerations as explained later. The result of this step is a set of context-automata that capture the activation of each service separately.

EXAMPLE 5.12 (GENERAL UNBUNDLING)
*We illustrate the general unbundling by integrating services F*04 *and F*08 *from the case study. We introduce basic context-automata for each service's context as shown in Figure 5.16. The labels are* 4_I *(entry condition of context* 4*),* 4_O *(exit condition of context* 4*),* 8_I *(entry condition for context* 8*) and* 8_O *(exit condition of context* 8*) to save space and enhance readability. The real conditions appear in use-cases UC*4 *and UC*8 *in Appendix B.2.*

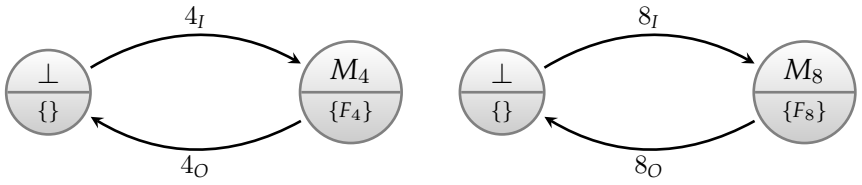


Figure 5.16.: The two basic context-automata for services *F*04 and *F*08

Integration of mode transition systems In the next step we integrate the context-automata into one mode transition system. Any mode transition system that is the result of an integration action can be integrated with yet another (basic or integrated) mode transition system. This allows an iterative application of the integration.

To describe the integration we define some auxiliary functions.

$$\Delta : M \to \mathcal{P}(\delta);$$
$$\Delta(m) \mapsto \{(m_i, (c, l), m_j) \in \delta \mid m_i = m\}$$

returns all outgoing transitions starting at a given mode. The auxiliary function

$$M_{out} : M \to (\mathbb{H}(C) \to \mathbb{B});$$
$$M_{out}(m) \mapsto \bigvee_{(m,(c_i,l_i),m_j)\in\Delta(m)} c_i$$

combines all conditions of all outgoing transitions of a mode.

Without restriction let $A_1 = (M_1, \delta_1, \Phi_1)$ and $A_2 = (M_2, \delta_2, \Phi_2)$ be two mode transition systems. The combined mode transition system $A = (M, \delta, \Phi)$ has elements

$$M = M_1 \times M_2$$

$$\delta = \{((m_1, m_2), (c, \rho), (n_1, m_2)) \mid (m_1, (c_1, \rho), n_1) \in \delta_1 \wedge c = (c_1 \wedge \neg M_{out}(m_2))\} \cup \quad (1)$$
$$\{((m_1, m_2), (c, \rho), (m_1, n_2)) \mid (m_2, (c_2, \rho), n_2) \in \delta_2 \wedge c = (c_2 \wedge \neg M_{out}(m_1))\} \cup \quad (2)$$
$$\{((m_1, m_2), (c, \rho), (n_1, n_2)) \mid (m_1, (c_1, \rho), n_1) \in \delta_1 \wedge (m_2, (c_2, \rho), n_2) \in \delta_2 \wedge c = (c_1 \wedge c_2)\} \quad (3)$$

$$\Phi = \{((m_1 \times m_2), (\Phi_1(m_1) \cup \Phi_2(m_2))) \mid (m_1 \times m_2) \in M_A\}$$

The transition relation captures three cases:

(1) Only mode transition system A_1 is able to make a transition

(2) Only mode transition system A_2 is able to make a transition

(3) both mode transition systems are able to make a transition

The integration preserves the switching behavior of both mode transition systems i.e. the parallel composition of the original two mode transition systems and the integrated mode transition system are bisimilar. The proof is a straight-forward application of Proof 18.1: we compose the two starting mode transition systems in parallel. Applying the distribution rule results in a nested mode transition system. It is easy to see that combining the respective mode-variables as a Cartesian product results in the set M. The sub-ordinate mode transition system appears as a sub-service in all modes of the super-ordinate mode transition system and hence is active all the time. Therefore, the history variables of both mode transition systems have the expected values.

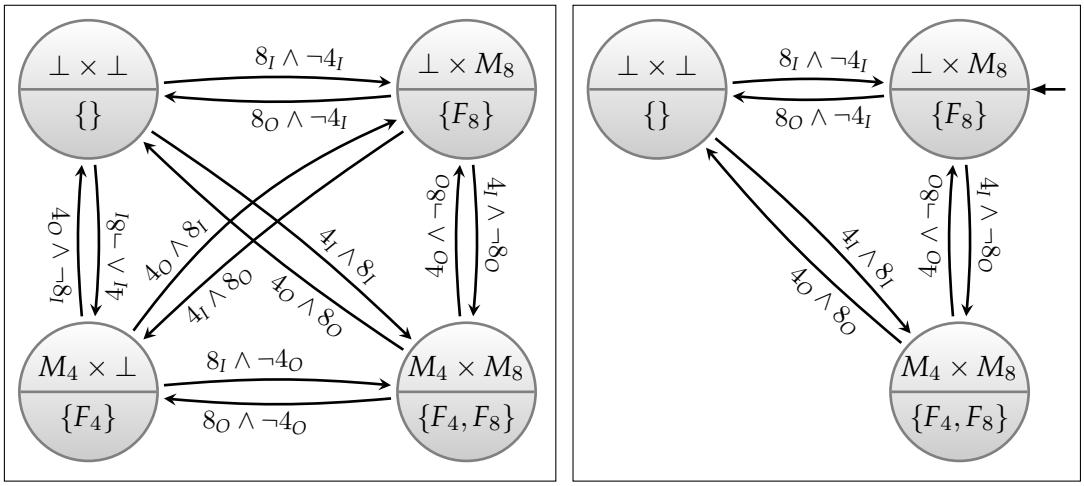
As a result of the integration the labels easily become complex; especially by integrating a large number of atomic automata. Therefore, after each iteration we apply a simplification.

1. we simplify labeling predicates using, e.g., the Quine-McCluskey algorithm [McCluskey, 1956] or the anti-link method [Beckert et al., 1998]

2. we remove any transition whose labeling predicate is not satisfiable

We keep modes without incoming transitions in this step. Such modes possibly are initial modes. Only after finishing the integration of all chosen services it is possible to accomplish a reachability analysis and drop any unreachable states that are not initial.

EXAMPLE 5.13 (GENERAL UNBUNDLING CONT.)
The integration of the basic context-automata results in the context-automaton on the left hand side of Figure 5.17a. We observe that $4_I \Rightarrow 8_I$, $8_O \Rightarrow 4_O$ *and* $4_I \wedge 8_O = FALSE$. *Using this knowledge we simplify the automaton: The labels* $(4_I \wedge \neg(8_I))$, $(8_O \wedge \neg(4_O))$ *and* $(4_I \wedge 8_O)$ *are unsatisfiable and we remove them. We are only going to integrate the two services* F_4 *and* F_8 *and choose* $\perp \times m_8$ *as the initial mode. Therefore, mode* $m_4 \times \perp$ *is unreachable and we remove it. Figure 5.17b illustrated the result.*



(a) Unbundling for services F_4 and F_8

(b) Optimization and determination of the initial mode

Figure 5.17.: Combination of the context-automata of services S_4 and S_8. mode-behaviors are dropped for brevity.

The example illustrates the need for the assignment of $\{\}$ *to the* $\perp$*-mode. The neutral nature of* $\{\}$ *with respect to set union allows a similar treatment than the combination of any other behavior in modes.*

In this example, the result is similar to the result of applying simple unbundling because the actual contexts of services F_4 *and* F_8 *are simple contexts.* ♣

Pre and post The processing of *pre*- and *post*-elements needs some additional attention. The *pre*- and *post*-elements refer to other contexts or even already elaborated modes (like in the work-flow example). This requires a careful handling of these elements and the names for modes while building the atomic context-automata.

The *pre*- and *post* conditions are additional predicates in the contexts of the requirements. We treat the *pre*- and *post* elements similar to other predicates during the combination of the transition guards in the integration step without considering their actual contents. The *pre*- and *post* elements are predicates that characterize sets. Therefore, it is possible to combine these elements during the integration step according to the set operators that correspond to the operators in logic:

$$pre = \mathcal{X} \wedge pre = \mathcal{Y} \equiv pre = \mathcal{X} \cap \mathcal{Y}$$
$$pre = \mathcal{X} \vee pre = \mathcal{Y} \equiv pre = \mathcal{X} \cup \mathcal{Y}$$
$$\neg pre = \mathcal{X} \equiv pre = \{y \in NAMES_{contexts} | x \notin \mathcal{X}\}$$

This minimizes the complexity of some transition labels.

During this final step we evaluate the *pre*- and *post* predicates. Any occurrence of a *pre*- or *post* element is replaced by a truth-value according to the following rules. Let $pre = \mathcal{X}$ and $post = \mathcal{Y}$ be sub-expressions of a transition label c in a transition $(m, (c, \rho), n)$.

$$pre = \mathcal{X} \models TRUE, \text{ iff } \exists\, m_i \in \mathcal{X} : m = (\dots, m_i, \dots)$$
$$post = \mathcal{Y} \models TRUE, \text{ iff } \exists\, n_i \in \mathcal{Y} : n = (\dots, n_i, \dots)$$

where $m = (m_1, \dots, m_r)$ and $n = (n_1, \dots, n_r)$ are modes that result from integrating r mode transition systems and the names of the integrated modes remain as part of the mode name. The creation and integration of original context-automata ensures that the mode names are different and can be distinguished now.

After replacing the elements by their truth value we evaluate and simplify the transition guards again. Some of them turn out to be unsatisfiable and we remove them. The result is a mode transition system whose transitions comply with the *pre*- and *post* elements of the original contexts.

EXAMPLE 5.14 (*Pre* AND *Post*)
We only consider a clipping of Example 5.7 to demonstrate the mechanism to remove transitions that do not satisfy the pre- and post conditions.

Figure 5.18 illustrates the aspired workflow with changes to the names of the modes to simplify the representation. We use the following context conditions:

$$context_{in}(A) \stackrel{def}{=} \top \wedge pre = \varnothing \qquad context_{out}(A) \stackrel{def}{=} c_1 \wedge post = \{B\}$$
$$context_{in}(B) \stackrel{def}{=} c_1 \wedge pre = \{A\} \qquad context_{out}(B) \stackrel{def}{=} (c_1 \vee c_2) \wedge post = \{C, D\})$$
$$context_{in}(C) \stackrel{def}{=} c_1 \wedge pre = \{B\} \qquad context_{out} \stackrel{def}{=} \bot$$
$$context_{in}(D) \stackrel{def}{=} c_2 \wedge pre = \{B\} \qquad context_{out} \stackrel{def}{=} \bot$$

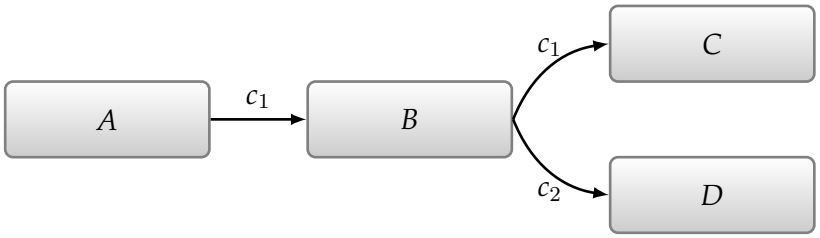


Figure 5.18.: The sequence of modes

By combining the contexts as described we create a number of modes and transitions. We can remove some transitions because they are not satisfiable. Others remain because we postpone the evaluation of the pre- and post elements. Figure 5.19 shows a small clipping of the resulting mode transition system. In addition to the expected workflow some additional modes and transitions exist which we highlight by red dashed lines in the figure.

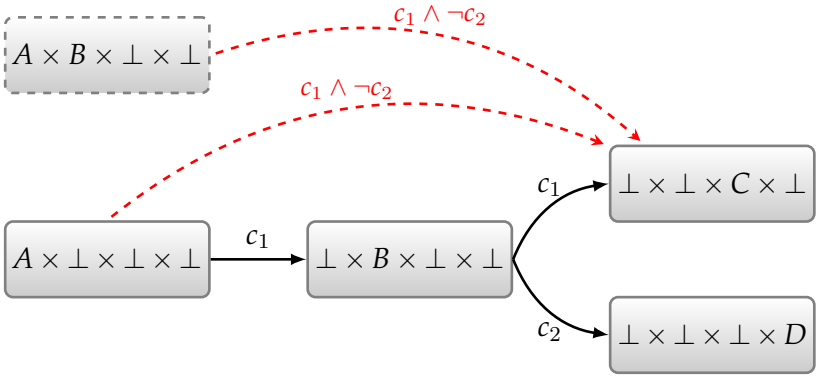


Figure 5.19.: A clipping from the mode transition system that results from the integration, yet without considering *pre-* and *post* elements

We focus on two transition relations which we are especially interested in.

1. *The transition from the mode* $(A \times \bot \times \bot \times \bot)$ *to* $(\bot \times \bot \times C \times \bot)$ *in general seems to be satisfiable because the exit and entry conditions without the pre- and post elements coincide. However, after evaluating the pre- and post elements, both become FALSE and by the conjunction, the complete transition guard becomes FALSE and we remove it.*

2. *The second transition we focus on is from* $(A \times B \times \bot \times \bot)$ *to* $(\bot \times \bot \times C \times \bot)$*. According to the integration step this transition is enabled if the exit conditions of the original contexts A and B hold at the same time together with the entry condition of the original context C. Without considering the pre- and post elements this is possible because all three conditions are* c_1*. However, the post-element of A contains B. The name of the original context B is not contained in the newly created mode name* $(\bot \times \bot \times C \times \bot)$*. Again the evaluation results in FALSE and the transition is removed.* ♣

The example illustrates the use of *pre* and *post* in the context of the workflow example (c.f. Example 5.7). However, creating work-flows by using the *pre*- and *post* elements this way is bulky. The methodological use of *pre*- and *post* elements MARLIN is only an extra constraint to certain transitions and is not intended to model full work flows. These constraints help modeling relations in the context like, e.g., "the engine must be started before driving".

If complex workflows are involved, modeling the work flow explicitly and transferring it to a mode transition system usually is easier than using *pre* and *post*. We refer to the formalism presented in [Leuxner et al., 2010] and [Leuxner et al., 2009b] for a more advanced method of designing workflows including more elaborated methods to express control flow splitting and joining.

Pursue and reset As an important note, we emphasis that the result of the approach is a preliminary mode transition system. The approach considers contexts from the contextual requirements chunks. However, there may be more aspects from different sources that are relevant for the mode transition system. These aspects include – among others – resuming, resetting, or continuing behaviors and the use of the *SAME*-service to ensure the integrity of variable values during the effectiveness of other modes.

Mode transition systems allow the mapping of long-term contexts by evaluating channel histories. They evaluate the contextual inputs starting at the last mode change together with any prior relevant inputs stored in the history variable.

Unlike Mealy/Moore automata [Mealy, 1955; Moore, 1956], the modes in mode transition systems (which roughly correspond to states) catch the history of transition-relevant-events only in an insufficiently way.

In Mealy/Moore automata, (control) states usually are an equivalence class for previous inputs and outputs. Activating an outgoing transition only depends on the current state and the current input. Therefore, for deciding about the activation of a transition, the path leading to this state is unimportant.

In mode transition systems this is different. Modes map behaviors but are no equivalence classes for input histories. The potential to satisfy some outgoing transition condition depends on the path that lead to the activation of the current mode. To be more precise, it depends on the history of inputs that lead to the mode. For two input histories, both leading to the same mode, upcoming inputs may be treated differently i.e. activate different outgoing transitions. Therefore, modes are no equivalence class of input histories. They are only an equivalence class of active behaviors. This is a tribute to the flexible handling of contexts and is best illustrated with an example.

EXAMPLE 5.15 (PRESERVATION OF CONTEXT HISTORY)
Assume a mode transition system that is created by integrating two independent services A and B whose entry conditions are defined as

$$A_{IN} = \{\!\{(\langle\bullet\rangle)^*\langle a\rangle\langle b\rangle(\langle\text{-}\rangle)^{[0-3]}\langle c\rangle\}\!\}$$
$$B_{IN} = \{\!\{(\langle\bullet\rangle)^*\langle\text{-}\rangle\langle c\rangle\ \ \langle\text{-}\rangle \qquad \langle e\rangle\}\!\}$$

Now, regard the following input stream s:

$$s = \langle\!\langle \ldots \langle a \rangle \langle b \rangle \langle - \rangle \langle c \rangle \langle - \rangle \langle e \rangle \rangle\!\rangle$$

After receiving the event $\langle c \rangle$, service A shall be activated. After receiving the event $\langle e \rangle$ service B shall be activated as well. Only a properly maintained history of events allows the consecutive activation of both services.

In a mode transition system that considers information only starting at the last mode change, B_{IN} is not satisfied in this setting because the sequence $\langle\!\langle \langle - \rangle \langle c \rangle \rangle\!\rangle$ is lost.

Using mode A as an equivalence class for received inputs fails as well. To illustrate this we create a mode transition system with a mode containing only A (we call it m_A), and one containing A and B (we call it m_{AB}). A transition leads from mode m_A to mode m_{AB} labelled with $\langle\!\langle \langle - \rangle \langle e \rangle \rangle\!\rangle$. The intention is to capture the fact that the prefix of the stream is already received. Now assume the input that leads to the activation of m_A is

$$s' = \langle\!\langle \ldots \langle a \rangle \langle b \rangle \langle c \rangle \rangle\!\rangle.$$

Receiving $\langle\!\langle \langle - \rangle \langle e \rangle \rangle\!\rangle$ subsequently shall not lead to the activation of service B but leads to mode m_{AB} where B is active. In this situation m_A is no proper equivalence class for earlier inputs because only some sequences satisfy A_{IN} and can be extended to satisfy B_{IN}. ♣

The example illustrates the existence of complex correlations between contexts – especially if contexts describe sequences of events – and explains the need for the history variable instead of using modes as an equivalence class for prefixes of conditions.

For simple unbundling, the reset and pursue flags have no effect because the transition guards only consider the last input anyhow. In general, it needs a good understanding of the system to decide about using the reset flag in a transition guard. A necessary condition for using the reset flag is:

$$\begin{aligned} \forall (m_i, (c_i, l_i), m), (m, (c_j, l_j), m_j) \in \delta, \\ w_1, w_1' \in L_{c_i}, \\ w_j \in L_{c_j} : \\ w_1 \circ \mathit{rest} = w_j \Rightarrow w_1' \circ \mathit{rest} \in L_{c_j} \end{aligned}$$

Informally spoken, all paths (i.e., I/O histories) leading to m are common prefixes of any path leading to some successor of m. Moreover, it is unimportant which path lead to m to decide about the transition from m to its successor. Basically, this requirement makes m an equivalence class for its incoming paths. In that case we can truncate the outgoing condition c_j by the common prefix and replace it with a condition c_i' that corresponds to *rest* in the above formula. Figure 5.20 illustrates this.

This is only a necessary but not a sufficient condition. We need to analyze any side effect to other transitions. A reset of a history variable possibly affects the transitions between other modes in the same mode transition system. To avoid this, we can use separate mode transition systems. The analysis and detection of prefixes to apply a

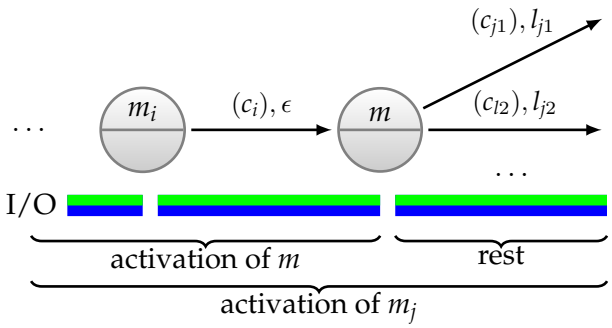


Figure 5.20.: The activation of m_2 must be a common prefix for all outgoing transitions of m_2

reset can be complex including their separation into classes of transitions that are independent to model them in different (nested) mode transition systems. However, this detailed analysis is out of scope of this thesis.

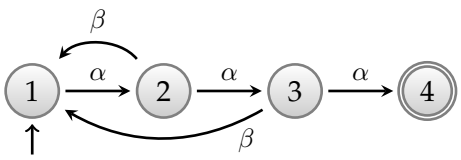
While the analysis for possible resets seems bulky from a formal point of view, the original description of the functionality may already contain useful information for deciding about resets. As an example take the protection mode of the trunk in the case study (UC 10).

EXAMPLE 5.16 (MODE TRANSITION SYSTEM WITH RESET FLAG)
Use-case 10 describes a setting where the system detects the opening gesture four times without a key at the right position and without starting the engine while in active mode. The system switches to the protection mode which prevents the leaching of the battery. If the engine starts while in protection mode the system switches back to the active mode.

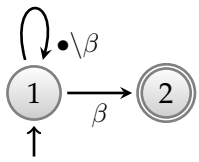
By using the reset flag we can easily describe the mode transition system. Let

$\alpha = kpos?away, TA?op$
$\beta = ER?start$ ♣

The transition guard from active to protection mode is a condition c_1 according to the DFA in Figure 5.21a. The condition c_2 guards the returning to normal operation according to the DFA in Figure 5.21b:



(a) Condition c_1 describing the switch to the protection mode



(b) Condition c_2 describing the switch to the active mode

Figure 5.21.: The conditions for the mode transition system using reset flags

Using these conditions we easily create the mode transition system in Figure 5.22 with the conditions only tracking I/O happening after the activation of a mode.

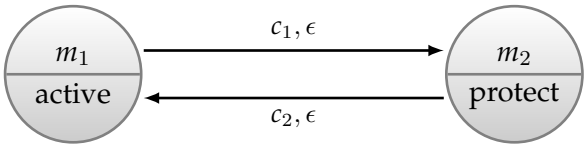


Figure 5.22.: The mode transition system for the protection mode

5.4.3. Combination of services by unbundling

The contextual requirements chunks are unspecific with the mode of combining services. They just list requirements and their contexts without making statements about their parallel or alternative composition. Therefore, we just joined the set of services during unbundling. The result of unbundling is therefore no final mode transition system but a preliminary one that needs further treatment of the mode behaviors: the designers must decide how to compose the services.

If the contextual requirements chunks are all either valid in parallel or alternative, one can exchange the join of services either by parallel composition or by alternative composition.

This observation allows a slightly different approach: before unbundling, the designers "sort" the contextual requirements chunks into groups such that each group describes the behavior of one logical function that will be composed in parallel with the functions that the other groups describe. Within the groups the contextual requirements chunks shall be alternatives.

After this sorting unbundling is applied stepwise:

1. Unbundling is applied to each of the groups separately. This separates alternatives that are available in common contexts from those that are available in different contexts. By replacing the service join with alternative composition during this step, one creates final mode transition systems that describe the behavior of distinct functions in different contexts.

   Simple unbundling During the step of "Creating disjoint equivalence classes", joining services

   $$SRVCS(C) \mapsto (SRVCS(A) \cup SRVCS(B))$$

   is replaced by alternative composing services

   $$SRVCS(C) \mapsto (SRVCS(A) \oplus SRVCS(B))$$

   in the cases "Overlapping", "Congruence", and "Subset" with a slightly changed auxiliary function

   $$SRVCS : NAMES_{CONTEXT} \to NAMES_{SERVICES}.$$

General unbundling During the construction of basic context automata we replace

$$\Phi = \{(m_\perp, \{\}), (m_n, \{f_n\})\}$$

with

$$\Phi = \{(m_\perp, FALSE), (m_n, f_n)\}.$$

Remember that *FALSE* is neutral with respect to alternative composition. During the integration of context automata we replace

$$\Phi = \{((m_1 \times m_2), (\Phi_1(m_1) \cup \Phi_2(m_2))) \mid (m_1 \times m_2) \in M_A\}$$

with

$$\Phi = \{((m_1 \times m_2), (\Phi_1(m_1) \oplus \Phi_2(m_2))) \mid (m_1 \times m_2) \in M_A\}.$$

2. After finishing unbundling of the groups we treat each of the resulting mode transition systems of the group as single services and apply unbundling between the groups. Now we create a mode transition system that combines the separate functions according to their general availability. Note that the actual (context adaptive) behavior of each of these functions is already handled by the mode transition systems that result from the first step.

   Simple unbundling During the step of "Creating disjoint equivalence classes", joining services

   $$SRVCS(C) \mapsto (SRVCS(A) \cup SRVCS(B))$$

   is replaced by parallel composing services

   $$SRVCS(C) \mapsto (SRVCS(A) \otimes SRVCS(B))$$

   in the cases "Overlapping", "Congruence", and "Subset" with a slightly changed auxiliary function

   $$SRVCS : NAMES_{CONTEXT} \rightarrow NAMES_{SERVICES}.$$

   General unbundling During the construction of basic context automata we replace

   $$\Phi = \{(m_\perp, \{\}), (m_n, \{f_n\})\}$$

   with

   $$\Phi = \{(m_\perp, TRUE), (m_n, f_n)\}.$$

   Remember that *TRUE* is neutral with respect to parallel composition. During the integration of context automata we replace

   $$\Phi = \{((m_1 \times m_2), (\Phi_1(m_1) \cup \Phi_2(m_2))) \mid (m_1 \times m_2) \in M_A\}$$

   with

   $$\Phi = \{((m_1 \times m_2), (\Phi_1(m_1) \otimes \Phi_2(m_2))) \mid (m_1 \times m_2) \in M_A\}.$$

This approach requires already some knowledge about the nature of the services. Basically, it is necessary to create a parallel normal form (cf. Definition 45) manually.

5.5. Embedding services into a development process

Service based specifications are useful for black-box functional descriptions of systems under development. Other artifacts in a development process like the architecture serve as specifications as well, but are less abstract. While a system architecture still has the characteristic of a specification it already has information about a logical distribution of functionality and captures design decisions (cf. the refinement chain, outlined in Figure 1.1). In this section we discuss an integrated use of service based specifications and component based descriptions of architectures that base on the close gearing of requirements engineering and architectural work.

5.5.1. Alternating application of service based and component based specification

Behavioral and structural refinement relates succeeding development artifacts [Broy, 2007]. Any successor in the chain of specifications has to be more specific about the allowed behaviors and constrains the allowed realizations by adding details about the structure.

In the development process of a system of considerable complexity the gap between the functional specification and the final architecture specification is significant. A hierarchical architecture design decomposes the functionality into interacting components at different levels of abstraction. This requires elaborating component requirements that are derived from the system specification such that composing the components satisfies the functional specification of the whole system.

The relation between the component requirements and the system requirements is non-trivial. An architecture handles additional information (e.g., non-functional requirements) that are not considered in the system specification, like, e.g., internal components and channels. As a consequence, a straight forward construction of the components' requirements is impossible and requires manual work. A methodological approach to elaborate this mapping of requirements is, e.g., presented in [Penzenstadler, 2011].

Harhurin proposes applying concepts of service based specification during developing an architecture [Harhurin, 2010]. The requirements of the components (and their respective formalization) at one level of a hierarchical architecture are functional specifications for the components at the next lower level. We repeat this idea because it fits well to the MARLIN methodology.

After describing the architecture at one level of abstraction we treat the newly defined components as black boxes and elaborate their requirements by applying the paradigm of service based specification. The idea that any component is a system as well [Broy and Stølen, 2001] justifies this step. Again we can apply the principles of behavioral analysis of Section 6 to ensure the consistency and completeness of component requirements. Finally we verify the component specification to comply with the system specification before continuing with the next step.

The iterative application of this principle supports a systematic development of a

system architecture in small incremental steps instead of an ad-hoc transition to a full architecture description. As a benefit, any of the incremental steps are analyzed and verified as valid refinements. Smaller increments hold a lower risk of introducing errors and support the commissioning of sub-systems. Figure 5.23 illustrates the alternating application of service based specifications and component specifications according to [Harhurin, 2010].

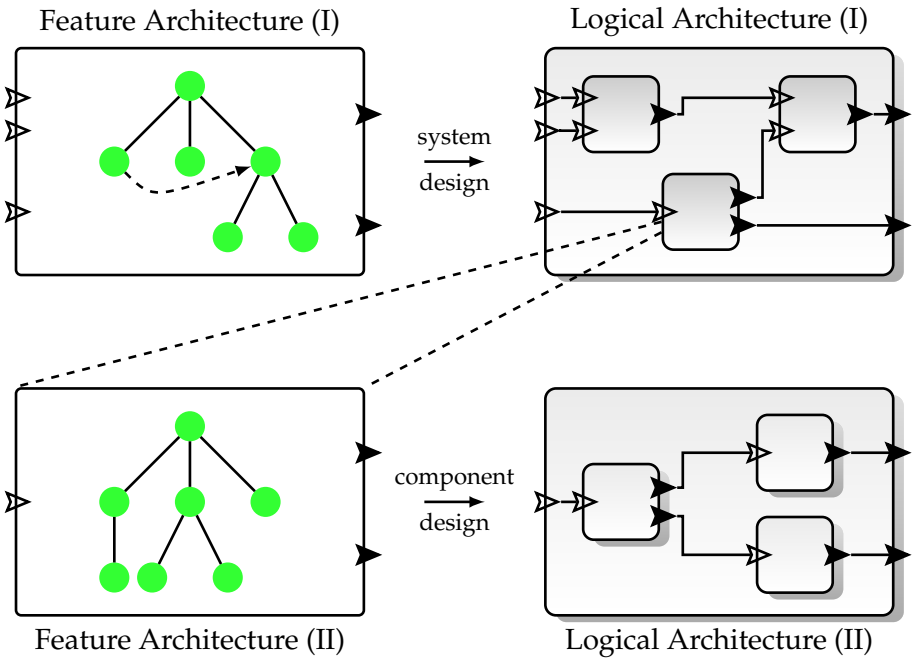


Figure 5.23.: Nesting of service specifications into a logical architecture

5.5.2. Cause-effect-chains

The project that is the basis of the case study of this thesis used the term cause-effect-chain to describe the interaction of components while processing signals from the sources to the sinks with respect to a single, user visible function. The term originally referred to projections of already existing component architectures only and had no precise definition. Within the project Base.XT Pro Live the idea was combined with the service based specification paradigm. The result is useful to relate service- and component-specifications. Cause-effect-chains work in two directions: constructive and analytically. We roughly outline both subsequently. A detailed consideration is out of the scope of this thesis and is still subject to more detailed investigations.

Constructive use We elaborate services as building blocks of the system just as described in this chapter. After fixing the behavior for the respective services, we elaborate components for each service that contribute to the service implementation.

After enhancing all service descriptions this way we integrate the services and their component extensions into a comprehensive component specification. Thereby, we identify congruent components of different services. Output channels of services that are no part of the system interface are feature interactions to other services. Input channels that are not part of the system interface represent influences from other services. We map the control of feature interactions to additional arbitration components. Finally, we connect the channels of the components.

Figures 5.24a and 5.24b illustrate two services A and B with their interface. Figures 5.24c and 5.24d show their component structures respectively. The internal channels are connected appropriately.

As a preliminary to this approach we need to reorganize the services. Conditional composition maps to a component structure by introducing additional components that implement the mode decision logic and communicate the result to affected components. Alternative composition cannot be directly mapped to a component structure. Therefore, we reorganize the service specification using the parallel normal form (PNF) (c.f. Definition 45 in Chapter 6). For short, the PNF extracts parallel services that are composed of alternatives.

Cause-effect-chains provide a methodological approach that allows a step by step definition of an architecture and the subsequent assembly of the generated fragments.

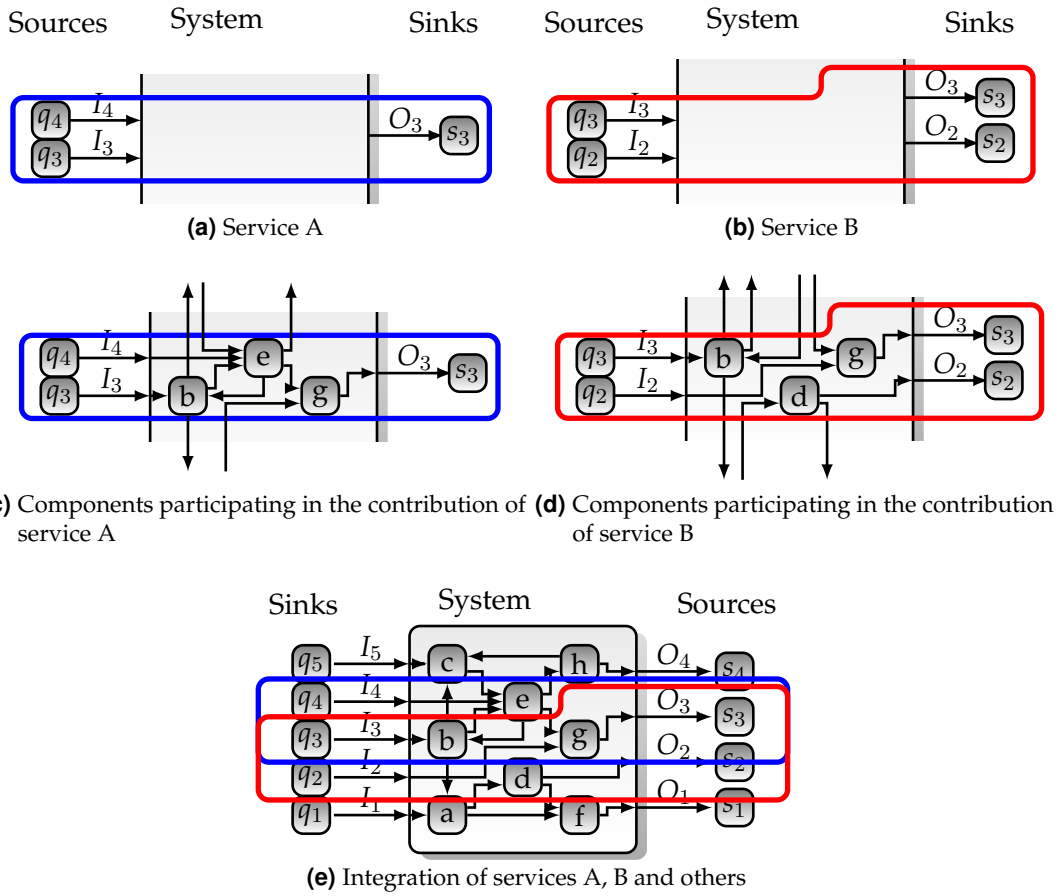
Analytic use For the change of an already specified and implemented service we are interested in information about the components that contribute to the service to make specific changes. Cause-effect-chains used analytically allow identifying sets of involved components.

This approach is due to an unpublished idea developed commonly with Doris Wild and Stefan Kugele. Determining the involved components bases on a bi-directional graph search. We regard component architectures as graphs with components as nodes and channels as directed arcs corresponding to the direction of the data flow. We formalize this as a tuple (V, E) with V as the set of nodes and $E \in V \times V$ as the edge relation between nodes. In addition we define the set of input channels as $I \in \bot \times V$ and the set of output channels $O \in V \times \bot$ such that input channels are edges without source and output channels are edges without target.

The elaboration of a cause-effect-chain requires three steps.

1. Starting at the set of output ports of the considered service we add all components recursively to the set of contributing components that are connected by arcs in inverse data flow direction. After termination, this set contains all components that contribute to the control of the respective output channels. This corresponds to a faithful slice of the system but potentially includes too much components. Formally, we choose the set o of output channels that are under consideration. We define

$$\mu S_m.\{v \in V | (v, \bot) \in o\} \cup \{v \in V | \exists v' \in S_m : (v, v') \in E\}$$



(a) Service A

(b) Service B

(c) Components participating in the contribution of service A

(d) Components participating in the contribution of service B

(e) Integration of services A, B and others

Figure 5.24.: Constructive use of cause-effect-chains to create architectures

2. In a second step, we execute a similar search starting at the input ports of the considered service. This time the search proceeds in data flow direction. The resulting set contains all components that are affected directly and indirectly by the input ports.

$$\mu S_n.\{v \in V | (\bot, v) \in i\} \cup \{v \in V | \exists v' \in S_n : (v,' v) \in E\}$$

3. In a final step, we intersect the two sets $S_m \cap S_n$. The resulting set contains all components that directly contribute to the service provision and are in the scope of the respective projection.

This analytical use of cause-effect-chains focuses on the input- and output channels of the service. As a result of the intersection of the results of the forward- and the backward search, the resulting components possibly have incoming and outgoing channels that do not correspond to channels at the system interface, nor are they connected to

one of the other components. These channels correspond to the feature interactions as demonstrated, e.g., in Figure 5.24c.

Note that cause-effect-chains resulting from this approach are possibly different to cause-effect-chains as used during the construction of the architecture. This relates to the difference of services as slices of an existing system and the constructive use of services as explained in Section 5.2.1. The analytical use extends the idea of slicing to architectures and allows identifying components that actively participate in providing a services.

5.6. Related work

The term service appears in different ways in literature. A term that is often used in a similar way is feature. The basic idea of a feature if used in a functional sense is to provide an early formalization of (functional) system requirements and to capture possible interactions between users and system [2]. Especially, features and services may describe a partial behavior. In the rest of this section we use the terms as introduced by the respective approaches that we discuss.

Zave and Jackson introduce features as functions that are accessible by a user (e.g., [Zave, 1993; Zave and Jackson, 2000; Jackson and Zave, 1998]). Features describe interaction patterns between users and the system under construction. They are "optional or incremental unit[s] of functionality" [Zave and Jackson, 2000]. This definition is close to use-cases [Bittner and Spence, 2002] but Zave and Jackson think of features as formal descriptions. They observe (unwanted) feature interactions after introducing new features and describe means to avoid them. They propose, e.g., role concepts, invariants [Zave, 1993] and an architecture called the Distributed Feature Composition (DFC) [Jackson and Zave, 1998] to address unwanted feature interactions.

Since the original work of Zave and Jackson focuses on the telecommunication domain their measures and their applications are tailored accordingly. An often cited scenario is combining selective call rejection and call forwarding. In this scenario it is unclear whether the phone system shall reject a call if a node forwards to a directory number whose rejection list contains the original caller but not the forwarding node. The DFC approach presents features as "boxes" through which a call is routed. By defining configurations of (compatible) boxes they clarify the specification [Jackson and Zave, 1998].

Zave and Jackson focus on introducing additional features that introduce new, possibly unwanted interactions. They are not concerned about contradictions of behaviors and do not consider the introduction of additional partiality. Furthermore, their solutions are specific for the telecommunication domain and include the use of structure and internal communication which is different for services as we regard them.

Schätz et al. [Schätz and Salzmann, 2003] develop an idea of services that we picked up in this thesis. Their characterization of services as underspecified clippings of behaviors is the basis of this thesis. They already formalize services as stream processing

[2] In non-functional approaches a feature can be anything like the color of a car, etc.

functions and outline the analysis for consistency and completeness. However, this early description of services includes no variables in the projection and offers only parallel composition. Schätz refines the idea of services and presents an approach that involves variable sharing, alternative, and parallel composition including the introduction of additional partiality in a later publication [Schätz, 2007, 2009]. We use these concepts for specifying the core system of context-adaptive systems and extend them by an explicit mode concept for specifying the adaptation sub-system.

In her thesis Rittmann presents a methodology for identifying and handling feature interactions [Rittmann, 2008]. She introduces a five-step methodology. After defining the basic services the approach identifies relationships between the services like *enable*, *disable*, *interrupt*, *continue*, *data dependencies*, etc. These services are subsequently integrated in a systematic way using a wrapping concept that captures the nature of feature interactions in automata and realizes the dependencies on additional inputs.

Deubler [Deubler, 2008] describes additional dependencies that may be picked up for the analysis of the application domain and the elicitation of the requirements. The modeling approaches of Rittman and Deubler are not directly applicable for modeling context adaptation but their methodological aspects are. Their proposed structured analysis of the application domain helps with identifying possible contradictions and feature interactions a priory and reduces the efforts for their analysis in the formalized specification.

In his PhD-thesis, Trapp presents a modeling approach for context-adaptive systems with a focus on dynamic reconfiguration. He uses feature models to describe system product-lines and enhances them to include information about reconfigurations. His features are close to the idea of our services. In his dynamic feature models Trapp focuses on the tree structure and dependencies between features like "requires" or "excludes" in the face of reconfigurations and omits the formal specification of leaves. He regards leaves as conventional features (modeled as non-adaptive systems). In Section 5.2.2 we use feature trees in a similar way but focus on the analysis of the modeled system specification. Therefore, we integrate the specification of the leaves with the specification of the dependencies and adaptations and present a holistic approach for specifying both.

Chapter 6

Analysis of specifications

A system specification is the basis of a development and often a part of a contract between customer and supplier. Therefore, a comprehensive specification is indispensable for a successful system development. While creating a specification, the designer collects and integrates the requirements of all stakeholders. For many reasons integrating requirements may result in deficiencies like contradictions or open aspects. Some of these deficiencies are obvious, others need an in-depth analysis to reveal them. Remaining deficiencies prevent proper realizations and cause a cost intensive re-engineering of the system if detected late in the life cycle [Boehm, 1981; Fairley, 1985] – not to speak of the risks that arise if safety related systems fail in use.

The system specification is the first (formalized) work product in a development and a correctness proof with respect to a predecessor is impossible. Besides validation of the requirements, an analysis for inconsistencies and partiality helps identifying inappropriate specifications. We discuss these properties in Section 6.1.

Subsequently we describe using the algebraic properties of MARLIN for separating the core system and the adaptation sub-system to support their analysis. We focus on the aspects of analyzing the context aware part of a system and the related properties in Section 6.2. In addition, we present an approach for preparing the core system for the analysis in Section 6.3.

We omit any practical aspects of executing the analysis and refer to the literature, e.g., [Baier and Katoen, 2008; Clarke et al., 1999; Leroy, 2010; Nipkow et al., 2002] etc. Section 6.4 presents related work for the analysis of (context-adaptive) systems.

Contents

6.1. Proof obligations

Techniques like model-checking analyze an actual implementation (or at least an abstract system model built from the implementation) for satisfying a number of formalized requirements (a specification) by proving (or disproving) the system representation being a logical model for the specification. The specification takes the role of a proof obligation for the implementation. Implementation and specification exist on different levels of abstraction.

No such proof obligation exists for formal specifications that were derived from informal requirements. Only the informal requirements are available for comparing the formal specification with the stakeholders needs. The informal character of initial requirements requires their checking by validation, e.g., through inspections and discussions or by prototyping and simulation rather then by fomal proof.

Zave et al. propose, among others, two *general proof obligations* for formal specifications that help with analyzing specifications [Gunter et al., 2002; Zave, 2001]. Loosely paraphrased they are "consistency" and "completeness". Although both are closely related in the context of MARLIN, methodologically we distinguish them. For our purposes we extend them with "context coverage" and "activation correctness".

Consistency We may think of a system specification as a set of axioms that characterize any proper implementation [Astesiano et al., 1999]. From mathematical logic we know that the consistency of axioms is a necessary condition for the existence of models. Similar, in model based development, consistency is a necessary condition for the existence of an implementation. Inconsistency is a consequence of conflicting requirements. Conflicts introduce *additional* partiality. Formalized as services, conflicting requirements require different values for commonly controlled resources like variables and output channels. Inconsistencies indicate serious problems that are sometimes caused by different needs of stakeholders. Therefore, conflicts need to be discussed with the stakeholders to find an agreement. It is dangerous to apply a causal chaos closure [Broy, 2005] to a specification that still has conflicts. The closure extends partiality with arbitrary behavior. As a result, the implementation satisfies neither of the original requirements.

Completeness A specification is complete if it defines (a set of) proper reactions for every possible input. An incomplete specification lacks definitions for system reactions for some inputs. There are yet unconsidered scenarios. An incomplete specification is unfinished. The interpretation of yet undefined behavior is a matter of the

semantic mapping (open world vs. closed world as discussed in Section 4.3). An according analysis reveals the unconsidered scenarios to supplement the missing interactions. This way a preliminary specification develops into a defined specification. Applying causal chaos closure to an incomplete specification is no problem if the stakeholders really have no detailed requirements for the input scenarios.

Context coverage For context-adaptive systems (and as a special case of completeness) we identify a third proof obligation. Some behavior has to be defined for all possible situations over the context model. Otherwise, a system may end up in "unknown situations" without properly defined behavior.

Activation correctness Finally, we investigate the activation correctness. The core system has only an informal counterpart in the contextual requirements chunks. In contrast, the contexts in the contextual requirements chunks are already formal. Unbundling ensures their correct mapping into a mode transition system. However, manually created mode transition systems must show their compliance in an analysis.

Incomplete and conflicting specifications as well as specifications with undefined scenarios are partial. Formally, a partial specification is unsatisfiable. We cannot assume that the environment uses only inputs that the system will accept [Gunter et al., 2002]. We better build a system that is able to react to all inputs in a proper way. Theorem 3 bases on a theorem of Broy [Broy, 2005] and supports this claim. The theorem and proof use the notion of strict causality from Section 2.4 but map this notion to the semantic domain of MARLIN in a straight forward manner.

THEOREM 3 (REALIZABLE SYSTEMS)
A service specification (in MARLIN*) either is input enabled*

$$\{(i,\sigma) \mid S.(i,\sigma) \neq \varnothing\} = \{(\vec{I}, \Sigma)\}$$

or does not contain any behavior

$$\{(i,\sigma) \mid S.(i,\sigma) \neq \varnothing\} = \varnothing$$

Note that the service must be able to react to every input in every state because any state might be present if the service is activated.

PROOF 3.1 (OF THEOREM 3):
The proof is by contradiction. Without restriction we assume a service specification F with

$$F.(\sigma, i_1) = \varnothing \text{ and } F.(\sigma, i_2) \neq \varnothing$$

for some $i_1 \neq i_2$*. By definition*

$$(i_1)_{\downarrow 0} = \langle\!\langle\rangle\!\rangle = (i_2)_{\downarrow 0}$$

With the definition of causality (which is now adapted to the semantic domain of MARLIN *in a straight forward manner):*

$$\forall i_1, i_2 \in \vec{I}, \sigma \in \Sigma, t \in \mathbb{N}:$$
$$(i_1)_{\downarrow t} = (i_2)_{\downarrow t} \Rightarrow \{(o)_{\downarrow t+1} \mid (o, \acute{\sigma}) \in F.(\sigma, i_1)\} = \{(o)_{\downarrow t+1} \mid (o, \acute{\sigma}) \in F.(\sigma, i_2)\}$$

and $t = 0$ *we conclude:*

$$\{(o)_{\downarrow 1} \mid (o, \acute{\sigma}) \in F.(\sigma, i_1)\} = \{(o)_{\downarrow 1} \mid (o, \acute{\sigma}) \in F.(\sigma, i_2)\}$$

This contradicts with the initial assumption where

$$\{(o)_{\downarrow 1} \mid (o, \acute{\sigma}) \in F.(\sigma, i_1)\} = \varnothing \qquad \textit{but}$$
$$\{(o)_{\downarrow 1} \mid (o, \acute{\sigma}) \in F.(\sigma, i_2)\} \neq \varnothing$$

q.e.d.

Another aspect of analyzing specifications are additional properties that constrain the acceptable implementations but are not directly mapped to I/O-behaviors. These properties are often safety properties. Examples are "it must never happen, that the trunk door opens while driving" or "eventually the service is resumed" (the last one is a liveness property). In functional safety, e.g., the safety goals take the role of such properties.

These properties are a "higher level specification" that can serve as a proof obligation for the specification under construction if they are formalized. The situation is then comparable to verifying implementations against specifications. MARLIN supports this kind of analysis with a small step semantics (cf. Section 6.3.3) that is close to an operational semantics and easily carries over to representations of specifications in model checkers like Mona [Klarlund and Møller, 2001].

As an alternative we can integrate the phrased properties into the specification. Specifications are collections of properties that can be extended with additional properties. Any temporal property can be transformed into an automaton [Vardi and Wolper, 1986]. Composing this automaton with an existing specification in parallel guarantees the desired property. The composition introduces an inconsistency if the specification does not satisfy the property. Therefore, checking a specification for satisfying an additional property is similar to composing the corresponding behavior in parallel and checking for inconsistencies. Subsequently, we focus on integrating the properties.

6.2. Analyzing the adaptation subsystem

MARLIN separates the core system and the adaptation sub-system by using mode transition systems (cf. Figure 5.3). This separation supports a separated analysis of both aspects of context-adaptive systems. We start with considerations about the adaptation sub-system.

The adaptation sub-system has two important properties. Depending on the approach of creating the mode transition systems both may be relevant for the analysis.

Correct activation and deactivation of behaviors Manually created mode transition systems can violate this property because transferring complex relations of contexts is error prone. Furthermore, manually created mode transition systems usually do not introduce a dedicated mode without behavior to model yet uncovered contexts. Because mode transition systems stay in a mode until an exit condition becomes true this may cause an inappropriate activation of some mode(s).

Contexts without defined behavior Unbundling ensures the correctness of service activation and deactivation by construction but the coverage is still an issue: the mode transition systems lack adequate mode behaviors if there exists no contextual requirements chunk that defines a behavior for a corresponding context. In that case the special mode $M_\perp$ is active.

To prepare the analysis for a correct activation of the adaptation-subsystem, we introduce a mode normal form that separates the core system and the adaptation subsystem even if the original specification mixes them. Furthermore, we demonstrate how to transform the adaptation-subsystem to make it accessible to, e.g., model checking approaches. The transformations are necessary to remove the history variable. The history variable prevents an effective execution of an analysis, because it is stream valued and therefore has an infinite type.

6.2.1. Context coverage

The context model defines the state space for possible contexts. Considering feature interactions and their influence on the observable behavior extend this state space. A specification needs to provide an adequate behavior for any (reachable) context within this state space. Causal chaos completion applied to a specification that still has contexts with undefined behavior allows arbitrary behavior in these contexts. It is unlikely that this matches any users' expectations.

We use a behavioral abstraction for the investigation of context coverage: we omit the actual formalization of the requirements that are active in a context and focus on the contexts themselves. The completeness of the behaviors is subject to Section 6.3. We illustrate the investigation of context coverage with an example.

EXAMPLE 6.1 (COVERAGE OF CONTEXT)
*We regard the two requirements R*04 *and R*08 *from the keyless entry case study. Their contexts are simple contexts* $\sharp(V = 0 \wedge \mathit{UBat} = \mathit{hi} \wedge \mathit{KPos} = \mathit{trunk})$ *and* $\sharp(V = 0 \wedge \mathit{KPos} = *\backslash \mathit{away})$. *Combining the two respective services covers a partition* $\sharp(V = 0 \wedge ((\mathit{UBat} = \mathit{hi} \wedge \mathit{KPos} = \mathit{trunk}) \vee (\mathit{KPos} = *\backslash \mathit{away})))$ *of the state space of the context model. We easily see that there is some behavior defined, e.g., for a current context* $V = 0 \wedge \mathit{KPos} = \mathit{DD}$ *but yet, there is no behavior defined for a current context* $V > 0$. ♣

We need to apply the analysis directly to the contexts in the contextual requirements chunks or a mode transition system resulting from unbundling.

Applying the analysis to mode transition systems that are manually created may be without use: a mode transition system is always in a certain mode. If the designer who creates a mode transition system is not already aware of uncovered contexts, it is unlikely that the mode transition system has a mode that represents yet uncovered contexts. In such case the mode transition system likely violates the (de)activation requirements of some services which will be revealed after the preparations in Section 6.2.3 that transfer mode transition system to make them accessible for an according analysis. However, despite discovering the deficiency, the transformed mode transition system offers no information about contexts without defined behavior.

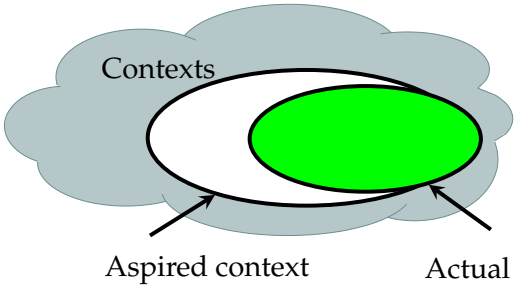


Figure 6.1.: Relation of state space defined over the context model, required activity of a combined service and the coverage of the services requirements.

Analysis of simple contexts We apply simple unbundling to *all* requirements that relate to a logical function under consideration to investigate their context coverage. Recall that simple contexts only refer to current inputs and define sets of valuations. The exit condition defines a complementary set.

We generate a set

$$CONTEXTS = \bigcup_{i \in CRC} context_{IN}(i)$$

The set *CONTEXTS* covers all contextual inputs with *some* defined behavior.

Let N be a set of aspects and $[N]$ be a context model mapping these aspects to types. The requirements cover the state space of the extended context model if their combined individual contexts cover this state space:

$$CONTEXTS \equiv \{v \mid \forall n \in N : v(n) \in [N](n_i)\}$$

We carry out this investigation relative to the context of the aspired logical function as illustrated in Figure 6.1 or, by including all requirements, relative to the complete system.

Analysis of general contexts Analyzing general contexts is more complex. For the general analysis of contexts we use the general unbundling and a reachability analysis of $\perp$-modes. It is sufficient to regard a simplified context-automaton $(\mathbb{M}, \delta)$ without maintaining the mapping of behaviors Φ since we are not interested in the actual behaviors.

A $\perp$-mode is a mode without defined behavior in the contextual requirements chunks. In the atomic context-automata we assigned *TRUE* to the mode-behavior of a $\perp$-mode because *TRUE* is neutral in parallel compositions and allows integrating additional requirements. The $\perp$-mode therefore is completely underspecified (non-determined) [1].

To identify uncovered contexts, we label the $\perp$-mode, and try to find paths to it. We shortly outline the procedure.

1. We start by labeling the mode $\perp$ of any atomic context-automaton with a predicate $bot(\perp)$.

2. The set $\mathbb{M}$ resulting from integrating two context-automata A and B is defined as $\{(m_i, m_j) \mid m_i \in \mathbb{M}_A \wedge m_j \in \mathbb{M}_B\}$. We label a state $(m_i, m_j) \in \mathbb{M}$ of the combined context-automaton $bot((m_i, m_j)) \Leftrightarrow bot(m_i) \wedge bot(m_j)$.

3. After unbundling all considered contexts we choose a (set of) initial modes IM.

4. We treat the simplified mode transition system as a graph G with nodes V and edges $A \in (V \times V)$. The coverage analysis is a query

   $$\exists\, i \in IM, j \in \mathbb{M} : (i, j) \in A^* \wedge bot(j)$$

   We do omit details of this query because well-established algorithms for graph search exist (see, e.g., [Heun, 2003]).

A reachable $\perp$-state indicates the existence of a sequence of events that lead to a context with no adequate behavior. Note that the analysis is independent from *TRUE* as a mode behavior. This allows intentionally under-specifying a behavior in a context by introducing a corresponding contextual requirements chunk with *TRUE* as a formalized requirement. This contextual requirements chunk appears as a named service in the specification and is different from $\perp$.

6.2.2. Mode normal form

In Appendix A.3 we present a number of algebraic properties that allow the transformation of specifications. Subsequently, we discuss the application of transformations with the goal to establish a useful normal form that separates the adaptation sub-system and

[1] Although the $\perp$-mode specifies no behavior in the contextual requirements chunks, in the mode transition system its mode-behavior must not be undefined (*FALSE*). This is a tribute to unbundling and the need for a neutral behavior with respect to parallel composition.

the core system for the analysis. The result is not unique, even not with respect to commutativity, but all possible results preserve the relevant properties and are equivalent with respect to the contained behavioral tuples.

The basis of all further considerations about the adaptation sub-system is the *mode normal form* (MNF). MARLIN-specifications can mix mode transition systems with parallel composition to provide a structure that matches the application domain. This abrogates the clear separation of core-system and adaptation sub-system from Section 3.1. The MNF restores this separation. The upper layers of the service hierarchy with their nested mode transition systems represent the decision process and the lower part with the composed services represents the core-system.

We start defining the mode normal with an auxiliary property called *free from modes*.

DEFINITION 39 (FREE FROM MODES):
We say that a service specification S is free from modes (FFM) if the service hierarchy of S does not contain a mode transition system at any level:

- *Any atomic service is free from modes.*
- *A compound service is free from modes if its sub-services are free from modes and the service is the result of either an alternative or a parallel composition.* □

Looking at a service's mode-variables easily allows to classify it as FFM: the set of mode-variables is empty for any service that is free from modes.

It is possible to transform services that are FFM into *parallel normal form* (PNF) and *alternative normal form* (ANF). Section 6.3 presents both normal forms.

DEFINITION 40 (MODE NORMAL FORM):
A specification is in mode normal form (MNF) if the only (nested) mode transition systems are at the top of the service hierarchy:

- *Any atomic service is in mode normal form*
- *Any compound service that is FFM is in MNF.*
- *A compound service is in MNF if it is a mode transition system and all of its mode-behaviors are in MNF.* □

By the following theorem and proof we support our claim that all context-adaptive systems have a separate core system and adaptation-subsystem.

THEOREM 4 (EXISTENCE OF A MODE NORMAL FORM)
For each service S there exists an equivalent service S' that is in MNF.

PROOF 4.1 (OF THEOREM 4):
Proof by induction on the structure of the specification.

BASE CASE: *If a formula S is an atomic service, S is already in MNF.*

INDUCTIVE STEP: *We consider the cases where S is either an alternative/parallel composition of services S_1 and S_2 or it is a mode transition system over services $S_1, \ldots, S_n$.*

Case 1: $S = S_1 \oplus S_2$
According to the requirement that alternative composition only can be applied to services whose set of mode-variables are empty the service S is free from modes. Hence, according to the definition, S is already in mode normal form.

Case 2: $S = S_1 \otimes S_2$.
According to the induction hypothesis there exist equivalent S'_1 and S'_2 in MNF. S is already in MNF if none of the S'_i is a mode transition system.
Without restrictions let $S_2 = (\{M_1, \ldots, M_n\}, \delta, \Phi)$ be a mode transition system. Hence,

$$S = S_1 \otimes (\{M_1, \ldots, M_n\}, \delta, \Phi)$$

such that $\Phi(M_i) = F_i$. We apply Theorem 18 i.e. the distributivity of parallel composition over mode transition systems and get

$$S' = (\{M_1, \ldots, M_n\}, \delta, \Phi')$$

such that $\Phi'(M_i) = F'_i = F_i \otimes S_1$. By the induction hypothesis there exist F''_i such that $F''_i \equiv F'_i$ and F''_i is in MNF. As a consequence

$$S'' = (\{M_1, \ldots, M_n\}, \delta, \Phi'')$$

with $\Phi''(M_i) = F''_i$ is in MNF.

Case 3: $S = (\{M_1, \ldots, M_n\}, \delta, \Phi)$. According to the induction hypothesis for all $F_i = \Phi(M_i)$ there exists $F'_i = F_i$ such that F'_i is in MNF. Trivially, by substitution of all F_i with F'_i the resulting service S' is in MNF. *q.e.d.*

Intuitively, for each path in a specification's service hierarchy that is in mode normal form, some node t exists such that all nodes on the path from the root to (excluding) t map to mode transition systems and t is a service that is free from modes. Figure 6.2 sketches the structure of a specification in MNF.

Let $((V, A, \phi), \Psi)$ be a service hierarchy in MNF. The node r is the root of the service hierarchy. If the service hierarchy is not FFM it holds that

$$
\begin{aligned}
&\forall (r,s) \in A^* : \mathit{leaf}(s) \Rightarrow \exists\, t : \\
&\quad (r,t) \in A^* \wedge (t,s) \in A^* && \wedge \\
&\quad \forall\, v \in (\mathit{path}(r,t) \backslash \{t\}) : \Psi(v) = (M_v, \delta_v, \Phi_v) && \wedge \\
&\quad \forall\, v \in \mathit{path}(t,s) : \Psi(v) = F_{v_1} \oplus F_{v_2} \vee \Psi(v) = F_{v_1} \otimes F_{v_2}
\end{aligned}
$$

The nodes t are no mode transition systems but are associated with the modes of mode transition systems i.e. they are mode-behaviors. Therefore, we call t terminal mode behaviors and the mode that is associated with this behavior a terminal mode. A terminal mode is part of a mode transition system and is associated with a service that is FFM. Figure 6.2 sketches the structure of a service hierarchy in MNF.

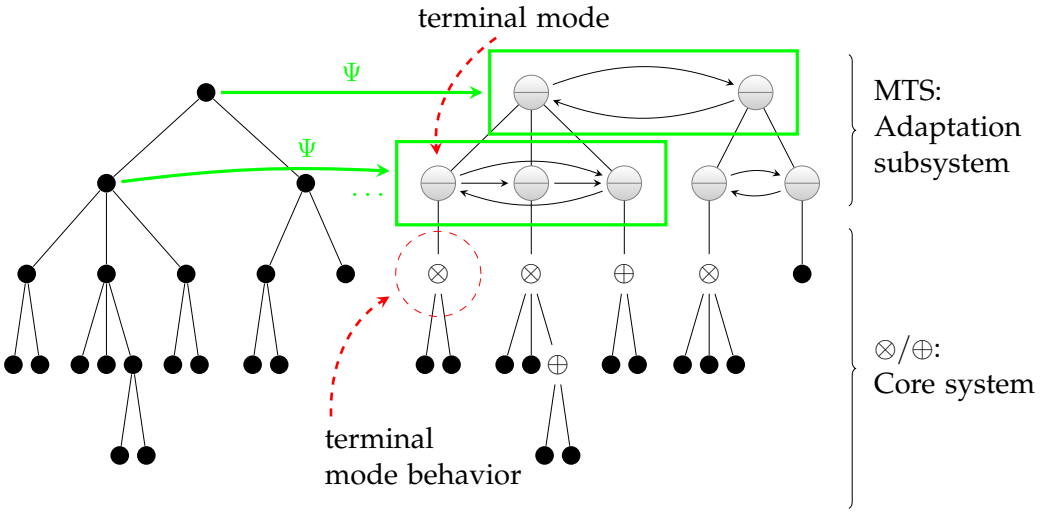


Figure 6.2.: Sketch of the structure of a specification in MNF

The mode normal form allows a limited sanity check of specifications to show unintended non-determinism. Mode transition systems allow arbitrary valuations for uncontrolled resources in a mode-behavior. Therefore, it is a good idea to check if all terminal mode behaviors have the same interface. This check is only possible in the MNF. In an arbitrary specification, some resource that is not controlled by a mode-behavior, may be controlled elsewhere. The MNF gathers all services such that no other service exists elsewhere that controls the resources at the same time. A simple syntactic check for interface coverage ensures that all terminal mode behaviors' interfaces coincide and no resources are uncontrolled in some mode.

6.2.3. Preparing mode transition systems for analysis

The mode normal form separates the behaviors of the core system from the adaptation sub-system to reduce the complexity of their analysis. Yet, the adaptation sub-system may be still too complex for the analysis. Although we deal with state transition diagrams that, in general, model checkers can handle, the history variables are stream valued. The infinite nature of these variables prevent an efficient analysis. The hierarchical structure is another obstacle for the analysis. Therefore, we describe structural transformations before passing the result to a verification method.

Mode transition systems allow modeling contexts of different complexity. We distinguish

- simple mode transition systems which only use simple contexts
- complex mode transition systems which use general contexts combined with reset flags and
- complex mode transition systems which use general contexts combined with pursue flags.

We do not discuss mixtures. Mixtures of simple contexts and general contexts generalize to mode transition system with general contexts. Combinations of reset and pursue flags are dangerous and shall not be used at all: if one transition relies on the complete history and another transition truncates the history variable, the resulting switching behavior is hard to analyze and usually does not match the intention of the pursue flags.

For the variants, different transformations with different complexity are available. We present them separately to allow applying the easiest one. The goal of any of the presented transformations is using structural properties to generate a finite transition system that allows an analysis by standard methods.

Note that the increasing complexity of the analysis is no matter of the complexity of MARLIN. MARLIN allows the modeling of complex systems. The applicable transformations are simple if a system is simple. If a system has complex mode changes, the transformations are complex. MARLIN only shifts the complexity from modeling to the analysis.

6.2.3.1. Removing the history variable for simple contexts or complex contexts with reset flags

The idea of this transformation is replacing the regular expressions at the transitions by actual modes. We systematically combine and replace the outgoing transitions of all modes.

Let $A = (\mathbb{M}, \delta_A, \Phi)$ be a mode transition system, $m \in \mathbb{M}$ be the mode under consideration, and δ_m the set of transitions $(m, (c_1, \epsilon), m'_1), \ldots, (m, (c_n, \epsilon), m'_n) \subseteq \delta$ starting in mode m. We create a mode transition system $A' = (\mathbb{M}', \delta'_A, \Phi')$ by the following transformation.

Step 1: If the transition labels of a transitions $(m, (c_i, \epsilon), m_i')$ are regular expressions we create deterministic finite automata (DFN) from the regular expression c_i, e.g., by applying the approach of Berry [Berry and Sethi, 1986]. Let this DFA be $(\mathcal{Q}_i, \Sigma, \delta_i, q_{0_i}, f_i)$ [2]. It is important to use new names for the set of states of the DFA that are not final, i.e. $(\mathcal{Q}_i \setminus \{f_i\}) \cap \mathbb{M} = \varnothing$. For transforming a mode transition $(m, (c_i, \epsilon), m_i')$ under consideration we choose $f_i = m_i'$, i.e., we ensure that the final state corresponds to the successor mode. Figure 6.3 illustrates an example mode transition system.

EXAMPLE 6.2 (REMOVING THE HISTORY VARIABLE FOR ANALYSIS)
We use a clipping of a hypothetical mode transition system to show the transformation that allows removing the history variable.

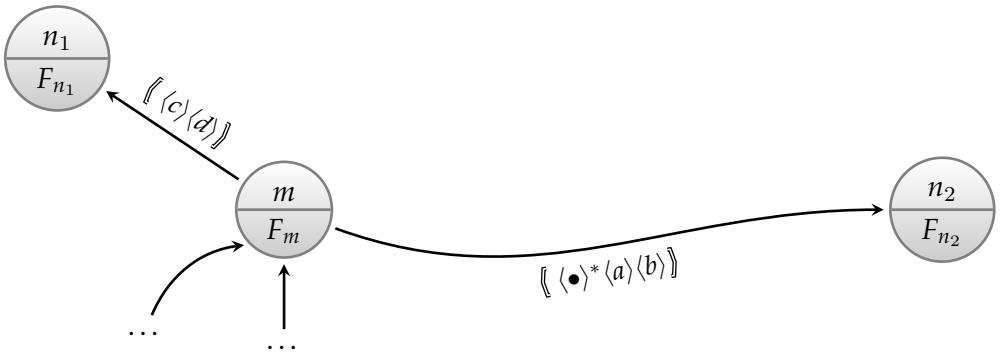


Figure 6.3.: Clipping of a mode transition system with two outgoing transitions

---

Recall that the input alphabet is $\Sigma = (c_1 \to T_1{}^*, \ldots, c_n \to T_n{}^*)$ i.e. it is a tuple of untimed streams representing the inputs at the relevant part of the input interface within *one* time interval. The alphabet Σ is the least upper bound of the alphabets of all transitions of a mode transition system. Later in this chapter we refer to these channels as C_{MTS}

Step 2: We repeat the transformation for every transition starting in m and get a set of automata $\{D_1, \ldots, D_n\}$. The mode transition system evaluates all transition labels at the same time. This corresponds to joining their corresponding languages. Therefore, we create a ϵ-NFA $N = (\mathcal{Q}_N, \Sigma, \delta_N, m, \mathcal{F}_N)$ that accepts any of the transitions with:

$$\mathcal{Q}_N = \bigcup_{i=1}^{n} \mathcal{Q}_i \cup \{m\}, \qquad \delta_N = \bigcup_{i=1}^{n} (\delta_i \cup \{(m, (\epsilon, \epsilon), q_{0_i})\}), \qquad \mathcal{F}_N = \bigcup_{i=1}^{n} \{f_i\}$$

[2] Note that we introduce a single final state rather than a set of final states. In MARLIN a mode transition system executes a mode transition immediately once a transition condition is satisfied. Therefore, outgoing transitions in a final state of a transition guard have no effect and there is no need to distinguish final states. Therefore, we can make use of a single accepting state and join states if a transition guard is given as a DFA already but has multiple final states

Note that we introduce m as a new initial state. This serves as a label for the final pasting operation. A ϵ-transition leads from m to the original initial states of each of the D_i. The final states differ since they correspond to different subsequent modes in the original mode transition system. The result of applying this step to the example of Figure 6.3 is in Figure 6.4

EXAMPLE 6.3 (REMOVING THE HISTORY VARIABLE FOR ANALYSIS (CONT.))
The transition conditions are transferred into a DFA and joined. The resulting ϵ-NFA accepts both conditions.

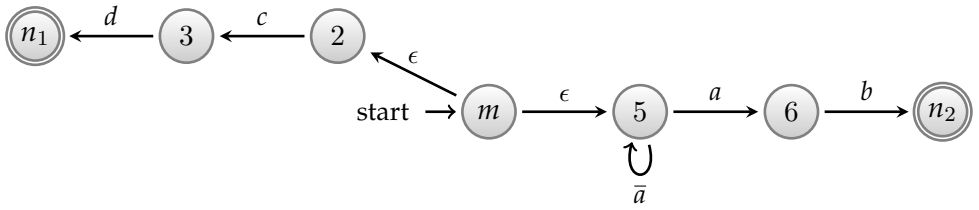


Figure 6.4.: Joining the transition automata

Step 3: Next we transform the nondeterministic automaton N into a deterministic variant D' by applying the powerset construction and an optimization if necessary to remove redundancy and unreachable states. During this optimization we need to exclude the final states from the optimization. This is necessary because we need to distinguish the final states. Without their separation we can only decide if a mode is left but lose the information about the successor.

We apply two more actions on the resulting DFA:

a) We remove any outgoing transitions of final states. In the original mode transition system a transition to a successor mode will put a new set of transition rules in place and reset the history. Therefore, it is impossible to continue after reaching a final state.

b) We introduce a new state $\perp_m$ by which we complete the automaton. Therefore, we extend the set of transitions in the following way and get a deterministic automaton D that is input complete except for final states.

$$\begin{aligned}\mathcal{Q}_D &= \mathcal{Q}_{D'} \cup \{\perp_m\} \\ \delta_D &= \delta_{D'} \cup \{(i,(\varsigma,\epsilon),\perp_m) \mid \nexists\, j \in \mathcal{Q}_D \backslash \mathcal{F}_{D'} : (i,(\varsigma,\epsilon),j) \in \delta_{D'}\} \\ &\quad \cup \{(\perp_m,(\varsigma,\epsilon),\perp_m) | \varsigma \in \Sigma\}\end{aligned}$$

The additional state is a trap and is reached if none of the original mode transition system's transitions can be satisfied any more. This is necessary to simulate the peculiarity of mode transition systems to stay in a mode until some outgoing transition is satisfied. The mode transition system is

stuck in the current mode if no outgoing transition can be satisfied any more [3]. After applying this step, the example looks like illustrated in Figure 6.5.

EXAMPLE 6.4 (REMOVING THE HISTORY VARIABLE FOR ANALYSIS (CONT.))
We optimize the ϵ-NFA and prepare it such that final states have no outgoing transitions and the trapping state makes the automaton input complete.

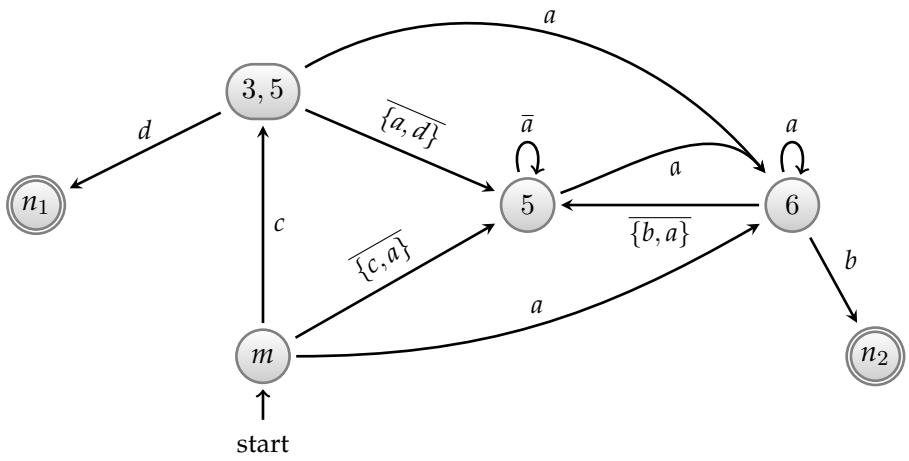


Figure 6.5.: Optimizing the transition automata

Step 4: We repeat the previous steps for all modes $m \in \mathbb{M}$ of the original mode transition system and get a couple of deterministic automata DA_m for each of them.

Now we paste these new automata DA_m according to the original mode transition system. To be more precise, we will introduce a new "local variable" as a part of the state names that records the progress of accepting the respective transition conditions of the original mode transition system.

a) We redefine the set of modes:

$$\mathbb{M}' = \bigcup_{m \in \mathbb{M}} (\mathcal{Q}_{DA_m} \backslash \mathcal{F}_{DA_m} \times \{m\})$$

Note that we are going to redirect the transitions to the final states of some DA_m to the state in $\mathbb{M}'$ that corresponds to the appropriate initial state of the original successor mode.

b) We change the transitions of the original mode transition system according to the DA_m while removing the original transitions.

$$\delta'_A = (\delta_A \backslash \bigcup_{i=1}^{n} \{(m, (c_i, \epsilon), m'_i)\})$$

[3] this scenario is not considered in the example: even if the transition labelled with $\langle\!\langle \langle c \rangle \langle d \rangle \rangle\!\rangle$ cannot be satisfied any more the other one labelled with $\langle\!\langle \langle \bullet \rangle \langle a \rangle \langle b \rangle \rangle\!\rangle$ may eventually be satisfied in the future

$$\cup \{((q,m),(c,\epsilon),(q',m))|(q,c,q') \in \delta_{DA_m} \wedge q' \notin \mathcal{F}_{DA_m}\} \tag{6.1}$$
$$\cup \{((q,m),(c,\epsilon),(m_i',m_i'))|(q,c,q') \in \delta_{DA_m} \wedge q' \in \mathcal{F}_{DA_m} \wedge m_i' = p\} \tag{6.2}$$

In (6.1) we add all transitions leading to intermediate states of the DFA. This includes the transitions starting in (m,m) (remember, we set the initial state of the ϵ-NFA to m). In addition, in (6.2) we "redirect" transitions that lead to final states in the DFA to states (m_i', m_i') that correspond to the successor modes of the accepting state in the DFA i.e. rather to switching to the final state of a DFA DA_m the transition system switches to the initial state of the successor mode's DFA.

c) We add mappings to Φ assigning the mode-behavior of mode m to all new modes.

$$\Phi' := \{((n,m),F)|(m,F) \in \Phi \wedge n \in \mathcal{Q}_{DA_m} \backslash \{\mathcal{F}_{DA_m}\})\}$$

Note that we carefully omitted elements in $\mathcal{F}_{DA_m}$ since they do not appear in the pasted mode transition system.

Figure 6.6 shows the result.

EXAMPLE 6.5 (REMOVING THE HISTORY VARIABLE FOR ANALYSIS (CONT.))
The automaton definition includes the mapping according to function Φ. We paste it into the original mode transition system such that the initial state relates to the original mode of the original mode transition system and the final states relate to the successor modes of the original mode transition system. Formally, we keep the history variable but it alway remains empty such that it has no more negative effect on the analysis and we can ignore it safely.

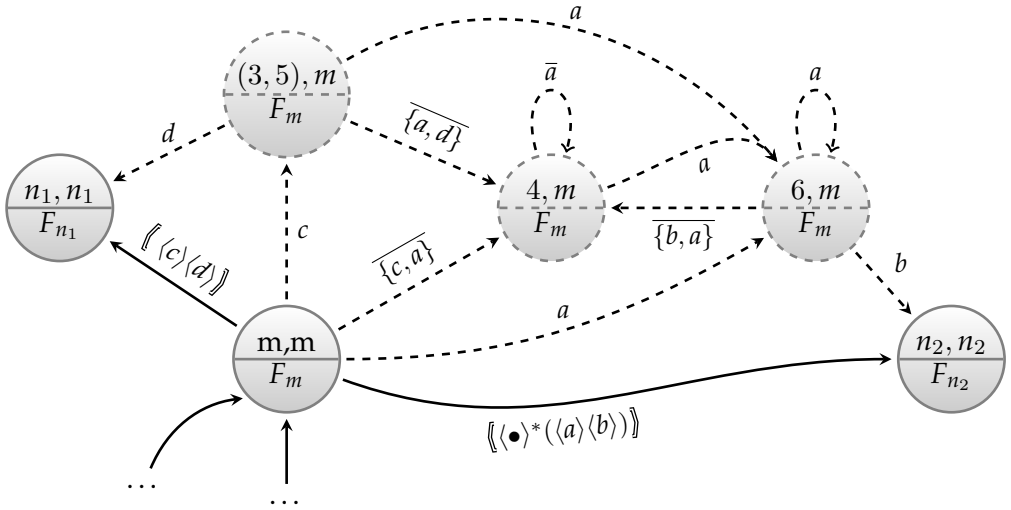


Figure 6.6.: Sketch of replacing a mode transition of complex contexts with simple contexts

Let $S = (\mathbb{M}, \delta, \Phi)$ be a mode transition system. The transformation creates a service that is "like" a mode transition system but executes a mode transition in every time interval. This allows ignoring the stream valued history variable.

To be suitable for any analysis we have to show that the resulting mode transition system corresponds to the original mode transition system in its relevant aspects. Therefore, we define an auxiliary function that allows us to talk about the active mode after receiving a sequence of contextual inputs over the input alphabet Σ.

DEFINITION 41 (SUCCESSORS OF MODES BY WORDS):
Let $S = (\mathbb{M}, \delta, \Phi)$ be a mode transition system. We define a function

$$succ : \{(\mathbb{M}, \delta, \Phi)\} \times (\mathbb{M} \times \Sigma^*) \times \Sigma^* \to \mathcal{P}(\mathbb{M} \times \Sigma^*)$$
$$succ_S((m,h),w) \mapsto \{(n,h') \mid (\sigma, i, o, \acute{\sigma}) \in [\![S]\!] \wedge i_{\lfloor C_S} = w \wedge \sigma.(PC_S) = m \wedge \sigma.(h_S) = h \wedge \acute{\sigma}.(PC_S) = n \wedge \sigma.(h_S) = h'\}$$

The function takes a mode transition system, the current state of the mode transition system (the values of the mode transition system variables) and a finite input stream as arguments and returns the state of the mode transition system (again the valuations of the mode transition system variables) after processing the stream. □

THEOREM 5 (CORRESPONDENCE OF MODE TRANSITION SYSTEMS)
Let $S = (\mathbb{M}, \delta, \Phi)$ be a mode transition system. After the transformation the resulting mode transition system $S' = (\mathbb{M}', \delta', \Phi')$ corresponds to the original mode transition system with respect to activating the mode-behaviors.

For a standard mode transition system the current mode and the history variable determine the switching behavior. Therefore, we define a binary relation $\mathcal{R} \subseteq (\mathbb{M} \times \mathbb{H}(C_S)) \times \mathbb{M}'$ (with C_S as the channels used for determining mode switches) that relates the mode and history variable of the original mode transition system to states of the transferred mode transition system without the history variable:

$$((m,h),n) \in \mathcal{R} \Rightarrow \Phi(m) = \Phi'(n) \wedge \forall w \in \Sigma^*, (m',h') \in succ_S((m,h),w)\ \exists (n', \langle\!\langle\rangle\!\rangle) \in succ_{S'}((n, \langle\!\langle\rangle\!\rangle), w) : ((m',h'),n') \in \mathcal{R}$$

This is a kind of simulation relation [Olderog, 1985; Bergstra and Klop, 1985] which includes the mapping to the mode-behaviors as an additional condition. We call a mode transition system S' a simulation of a mode transition system S iff

$$\forall m \in \mathbb{M}, h \in \Sigma^*\ \exists n \in \mathbb{M}' : ((m,h),n) \in \mathcal{R}$$

and claim that every S' that is the result of applying the above transformation to a mode transition system S is a simulation for S.

PROOF 5.1 (OF THEOREM 5):
Proof by induction over the contextual input w. We define a relation $\mathcal{R}$ and proof its relevant properties. Let $DA_m = (\mathcal{Q}_m, \Sigma, \delta_m, q_{0_m}, \mathcal{F}_m)$ be the DFA that accepts all the transition conditions starting in some mode m according to the transformation up to step 3.

$$\begin{aligned}\mathcal{R} \stackrel{def}{=} &\{((m, \langle\!\langle\rangle\!\rangle), (n, m)) \mid m \in \mathbb{M} \wedge n = q_{0_m}\} \cup \\ &\{((m, h), (n, m)) \mid m \in \mathbb{M} \wedge (q_{0_m}, h, n) \in \delta_m^*\}\end{aligned}$$

By the definition of reset we know that any h occurring in this relation has no prefix that satisfies a transition in the mode transition system S. If there was such a prefix, the transition would have been taken. In this case the successor of the mode m and the rest of the input (truncated by the prefix) are in the relation.

BASE CASE:

$$\nexists(m, (c, \epsilon), m') \in \delta, w_1 \in \Sigma^* : w_1 \sqsubseteq h \circ_t w \Rightarrow w_1 \in L_c$$

Informally spoken, there is no prefix of w that begins with h and is sufficient to satisfy some transition condition. At this point we distinguish two cases:

1. *None of the outgoing transition will ever accept any extension of w:*

   $$\nexists(m, (c, \epsilon), m') \in \delta, w' \in \Sigma^* : h \circ_t w \circ_t w' \in L_c$$

   For this case we extended the transferred mode transition system with transitions to a trapping state in step 3b. Both, the original and the transferred mode transition system stay in a mode where the original mode behavior is active. Formally,

   $$\begin{aligned}&((m, h), (n, m)) \in \mathcal{R}, \\ &succ_S((m, h), w) = \{(m, h \circ_t w)\}, \qquad succ_{S'}(((n, m), \langle\!\langle\rangle\!\rangle), w) = \{((\bot_m, m), \langle\!\langle\rangle\!\rangle)\}\end{aligned}$$

   with $(q_{0_m}, h, n) \in \delta_m^*$ and $(n, w, \bot_m) \in \delta_m^*$ also $(q_{0_m}, h \circ_t w, \bot_M) \in \delta_m^*$. Therefore,

   $$((m, h \circ_t w), (\bot_m, m)) \in \mathcal{R} \qquad \Phi(m) = \Phi'((\bot_m, m))$$

   Note, if already h is an input that cannot be extended such that it will finally be accepted by a transition condition, initially, $n = \bot_m$ already. In this case, S′ stays in the trapping state while reading w.

2. *The input w is a prefix to an input that is possibly accepted by some transition condition c in the future:*

   $$\exists(m, (c, \epsilon), m') \in \delta, w' \in \Sigma^* : h \circ_t w \circ_t w' \in L_c$$

In that case w will lead to an intermediate state of the transition automaton and hence to some mode (n'', m) that maps to the appropriate mode behavior and is able to accept further inputs. Formally,

$((m,h),(n,m)) \in \mathcal{R}$,
$succ_S((m,h),w) = \{(m, h \circ_t w)\}, \qquad succ_{S'}(((n,m),\langle\!\langle\rangle\!\rangle),w) = \{((n'',m),\langle\!\langle\rangle\!\rangle)\}$

with $(q_{0_m}, h, n) \in \delta_m^*$ and $(n, w, n'') \in \delta_m^*$ also $(q_{0_m}, h \circ_t w, n'') \in \delta_m^*$.
Therefore,

$$((m, h \circ_t w),(n'',m)) \in \mathcal{R}, \qquad \Phi(m) = \Phi'((n'',m))$$

All states of $\mathbb{M}'$ that correspond to intermediate states of D_m relate to the same mode-behavior as the mode under consideration of the original mode transition system. The original mode transition system has not left its mode yet.

INDUCTION HYPOTHESIS:

$\forall w \in \Sigma^* : ((m,h),(n,m)) \in \mathcal{R} \Rightarrow ((m',h'),(n',m')) \in \mathcal{R}$
with $(m',h') \in succ_S((m,h),w)), ((n',m'),\langle\!\langle\rangle\!\rangle) \in succ_{S'}(((n,m),\langle\!\langle\rangle\!\rangle),w)$

INDUCTION STEP:

$\exists (m,(c,\epsilon),m') \in \delta, w_1 \in \Sigma^* : w_1 \sqsubseteq h \circ_t w \wedge w_1 \in L_c$

Informally spoken, a prefix of w is (together with the history saved in h) sufficient to satisfy some transition condition. Then there exists a w' such that $h \circ_t w' = w_1$ i.e. w' is the prefix of w that leads to the activation of a transition. Furthermore, there is a w'' such that $w_1 \circ_t w'' = h \circ_t w$ i.e. it is the suffix of w that is left after truncating w'. Formally,

$((m,h),(n,m)) \in \mathcal{R}$,
$succ_S((m,h),w) = \{(m',h')\}, \qquad succ_{S'}(((n,m),\langle\!\langle\rangle\!\rangle),w) = \{((n',m'),\langle\!\langle\rangle\!\rangle)\}$

Let m' be such that $(m,(c,\epsilon),m'') \in \delta$ is the transition whose condition c is satisfied by $h \circ_t w'$. Therefore, after activating the transition, the original mode transition system is in a state $(m', \langle\!\langle\rangle\!\rangle)$.

With $(q_{0_m}, h, n) \in \delta_m^*$ and $(n, w', m') \in \delta_m^*$ also $(q_{0_m}, h \circ_t w', m') \in \delta_m^*$ with $m' \in \mathcal{F}_m$. Since we removed all outgoing transitions of any final state in DA_m we also know that w' is the shortest input with this property.

From the induction hypothesis we know that $((m',\langle\!\langle\rangle\!\rangle),(m',m')) \in \mathcal{R}$. (m',m') is the state in $\mathbb{M}'$ that corresponds to the final state in $\mathcal{F}_m$ and has the name m' of the successor mode in $\mathbb{M}$ as a name constituent. This mode relates to the same mode-behavior as m'. According to the induction hypothesis $(m',h') \in succ_S((m',\langle\!\langle\rangle\!\rangle),w')$ and $((n',m'),\langle\!\langle\rangle\!\rangle) \in succ_{S'}(((m',m'),\langle\!\langle\rangle\!\rangle),w')$. Therefore we conclude

$$((m',h'),(n',m')) \in \mathcal{R}, \qquad \Phi(m') = \Phi'(n') \qquad \text{q.e.d.}$$

The use of reset flags in mode transitions enforces modes to be equivalence classes for their activation (see also the comment in Section 5.4.2). The history that leads to a mode has no influence on the outgoing transitions of that mode in mode transition systems with reset flags in all transitions. This eases their preparation for analysis because we can consider the outgoing transitions apart from any other mode's outgoing transitions.

6.2.3.2. Removing the history variable for complex contexts with pursue flags

Again our goal is to remove the history variable. However, the preparation of the analysis becomes more complex for mode transition systems with pursue flags. In contrast to mode transition systems with reset flags, inputs that occurred long before the current mode was activated influence the mode transitions. Hence, the mode transition system evaluates all mode transition conditions of all modes concurrently but only enables those transitions starting in the current mode.

Let $(\mathbb{M}, \delta, \Phi)$ be a mode transition system. An example is in Figure 6.7. Again we transfer the mode transition system in separate steps.

Step 1: We start with transforming the transition labels into DFAs. This time we execute this for all transition labels at once. As a result for any transition $(m, (c, \rho), m') \in \delta$ we get a DFA $D_{m_i} = (\mathcal{Q}_{m_i}, \Sigma, \delta_{m_i}, q_{0_{m_i}}, \mathcal{F}_{m_i})$ where $0 < i \leq n_m$ with $n_m = \mid \{(i, (c, \rho), j) \mid i = m\}$ as the number of transitions starting in mode m.

EXAMPLE 6.6 (REMOVING THE HISTORY VARIABLE IN RESUMING MTS)
We use an example inspired by the case study. We simplify it to focus on the peculiarities (rather than five detected misuses in a row we use only one and we use abstract events only) and modify it to include a possible repair.

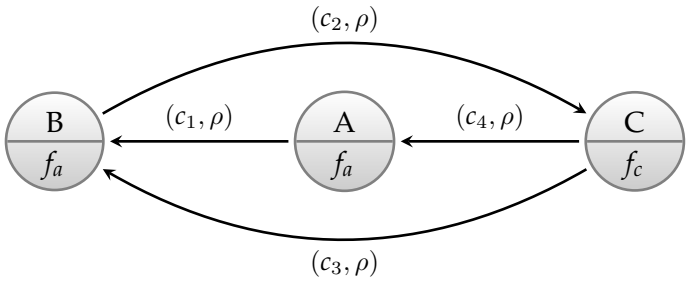
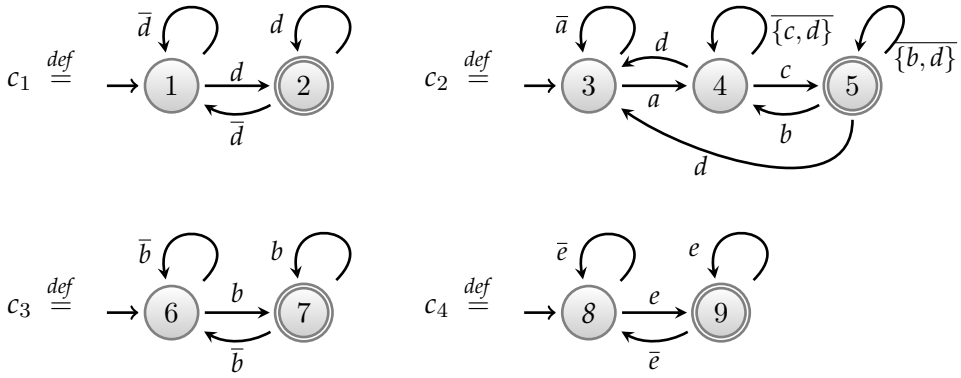


Figure 6.7.: Example of an MTS with pursue flags

In mode B the gesture detection works as described. A is a mode used for maintenance and C is the mode with an active protection and without gesture detection. Event a indicates the first incident causing the system to be more "attentive" and c represents the second incident causing the deactivation of the gesture detection by switching to mode C.

We also introduce three curing events b,e, and d with the following meaning in the case study:

- *b indicates an event that resets the misuse counter (e.g., starting the engine). This event allows the reactivation of the gesture detection but only transfers the control to the "attentive" state because there might still something be wrong.*
- *d indicates a repair of the system. Now the system returns to normal operation in mode B because the system is tested and possibly repaired.*
- *e indicates the start of a maintenance.* ♣



We deal with the transition automata in a similar way as described in Section 6.2.3.1 up to step 3 but with some differences:

- we omit step 3a where we remove outgoing transitions of final states. Remember that step 3a is necessary in reset mode transition systems because continuing in a final state is impossible. Now we deal with pursue. Hence, we want to be able to continue evaluations.
- we do not use the names of the destination modes as the names for the final states but insist in using fresh names for every state of each D_m.
- We maintain an extra relation that records the destinations of the respective final states, i.e. $\mathcal{M}_m \subseteq \mathcal{F}_m \times \mathbb{M}$

As a result of applying these steps, we get DFAs $DA_m = (\mathcal{Q}_m, \Sigma, \delta_m, q_{0_m}, \mathcal{F}_m, \mathcal{M}_m)$ for all $m \in \mathbb{M}$ that possibly have different final states (if there were multiple outgoing transitions) and the following properties:

- $\mathcal{M}_m(q) = m'$ iff $\exists\, \mathcal{F}_{m_i} : q \in \mathcal{F}_{m_i} \wedge (m, (c_i, \rho), m') \in \delta$ where $\mathcal{F}_{m_i}$ is the set of final states of a transition automaton DA_{m_i} that is the result of treating the transition $(m, (c_i, \rho), m')$ i.e. the elements in $\mathcal{M}_m$ relate final states to the destination mode of the transitions in the original transition automata starting in mode m before their integration.
- $\forall\, m, n \in \mathbb{M} : m \neq n \Rightarrow \mathcal{F}_m \cap \mathcal{F}_n = \varnothing$, i.e., the final states of all transition automata starting in different modes are disjoint.

Furthermore, the DA_m are now extended with the trapping state and therefore input complete. Note that this is true for the final states as well.

EXAMPLE 6.7 (REMOVING THE HISTORY VARIABLE IN RESUMING MTS)
In the example we only consider c_3 and c_4. Those transitions have the same source mode and serve well as examples. However, they do not include the trapping state because they are already input complete.

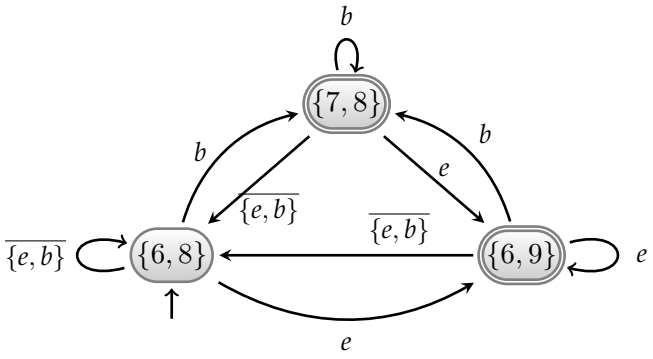


Figure 6.8.: The combination of transition automata c_3 and c_4

Step 2: In the next step we integrate the transition automata DA_m. We record their state with respect to the inputs regardless if their source mode is active. Since all of them are input complete, we can do this by parallel composition without losing transitions[4]. Formally we create an auxiliary automaton $D_{TR} = (\mathcal{Q}_{TR}, \Sigma, \delta_{TR})$ representing all transition automata with the Cartesian product of their respective set of states and executing the respective transitions simultaneously.

$$\mathcal{Q}_{TR} = \prod_{m=1}^{n} \mathcal{Q}_m \qquad \delta_{TR} = \{((\bigtimes_{m=1}^{n} q_{i_m}), \varsigma, (\bigtimes_{m=1}^{n} q_{j_m})) \mid \exists_{m=1}^{n} (q_{i_m}, \varsigma, q_{j_m}) \in \delta_m\}$$

Subsequently we will use the notation $q_{i_m} \in q_i$ to indicate $q_i = (\dots, q_{i_m}, \dots)$

EXAMPLE 6.8 (REMOVING THE HISTORY VARIABLE (CONT.))
We integrate the transition automaton for c_2 and c_3 but ignore c_4.

[4] If they were not input complete, certain transitions could disappear if some DA_m is inactive. This leads to situations, where the original mode transition system can react (because the transition in question has not yet an influence and possibly never will have) but the transformed one lacks a transition and blocks. With the transition automata extended with a trapping state the situation changes such that the DA_m in question switches to this trapping state and will not block.

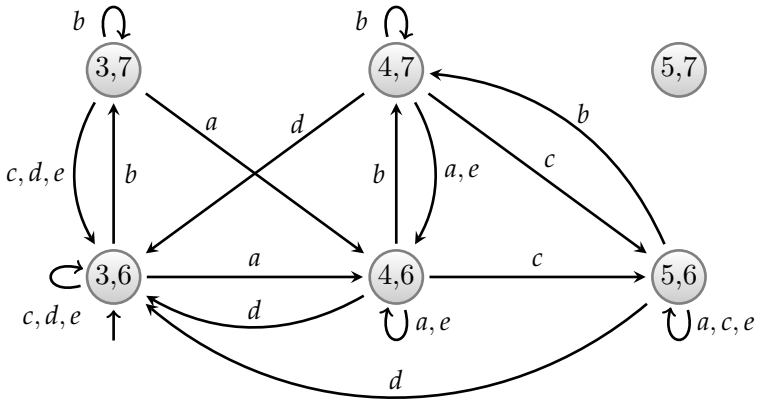


Figure 6.9.: The product automaton of c_2 and c_3

Step 3: To this end, the resulting automaton tracks the state of any transition automaton. As a next step, we add capabilities to track the active mode. We start this by creating an automaton $D = (\mathcal{Q}_D, \Sigma, \delta_D, q_{0_D}, \mathcal{F}_D)$

$$\mathcal{Q}_D \subseteq \mathcal{Q}_{TR} \times \mathbb{M}$$
$$\mathcal{F}_D = \{(q, m) \in \mathcal{Q}_D \mid \exists (p, m') \in \mathcal{M}_m : p \in q\}$$
$$q_{0_D} = \bigcup_{m \in \mathbb{M}} (\bigtimes_{m=1}^{n} q_{0_m}) \times m$$
$$\delta_M = \{((q_i, m), (\varsigma), (q_j, m)) \mid ((q_i), (\varsigma), (q_j)) \in \delta_{TR} \wedge (q_i, m) \notin \mathcal{F}_D\} \cup \tag{6.3}$$
$$\{((q_i, m), (\epsilon), (q_i, m')) \mid \exists (q, m') \in \mathcal{M}_m : q \in q_i\} \tag{6.4}$$

Informally spoken we make the following steps:

- We extend the set of states by a label that indicates the active mode.
- We choose the final states to correspond to the final states of the transition automata F_m if the flag of the active mode is m.
- We create the set of transitions such that any transition of δ_{TR} is applied to any combination of a state $q_i \in \mathcal{Q}_{TR}$ with a flag m for the active mode, if its source is no final state (c.f. (6.3)).
- In addition we introduce transitions that lead from the final states (which indicate the accepting of an input in an active mode) to their pendant that corresponds to the next active mode (c.f. (6.4)). Therefore, we choose the target $(q_i \times m')$ such that the states of the transition automata are equal and the new active mode corresponds to the original follow-up mode.

EXAMPLE 6.9 (REMOVING THE HISTORY VARIABLE (CONT.))
Basically, the operations in this step instantiate the automaton D_{TR} for each mode and add the transitions accordingly: as long as no transition is accomplished in the original mode transition system the transitions in the created automaton stay within one "instance".

The new automaton executes an epsilon transition to the instance that corresponds to the successor mode in the original mode transition system. The information for accepting the inputs of all instances remains the same.

In Illustration 6.10 we highlight transitions of D_{TR} that are not used because a mode transition executes first with red dashed lines. Those transitions are not contained in the transition relation δ_M although they are contained in the transition relation δ_{TR}. The transitions that correspond to mode transitions in the original mode transition system are green. As we can see, some transitions are exclusive to one "instance" and other

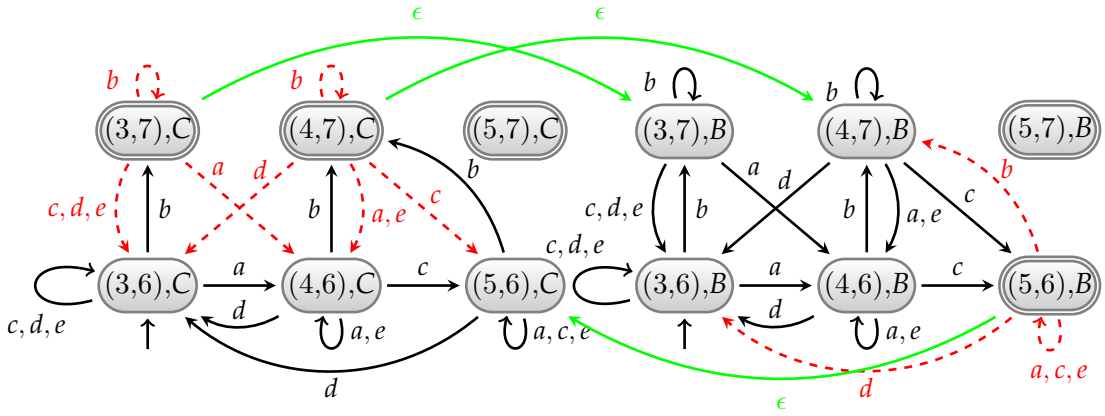


Figure 6.10.: Linking the instances of B and C together. Outgoing transitions of final states (red and dashed) are replaced by ϵ-transitions to their pendant (green)

transitions are present in both "instances". Take note of the transitions $(36C, b, 3BC)$ and $(36B, b, 3BB)$. The second one reads a "b" but unlike the first one does not lead to the change of the mode although both descend from the same transition in a transition automaton. This is because mode C is inactive. ♣

Step 4: Finally we create the transformed mode transition system $M' = (\mathbb{M}', \delta', \Phi')$ that changes its mode for every input and therefore can be analyzed without the history variable.

$$\mathbb{M}' = \mathcal{Q}_D \qquad \Phi' = \{((q, m), \Phi(m))\}$$
$$\delta' = \{(q, (\varsigma, \epsilon), q') \mid (q, \varsigma, q') \in \delta_D \wedge q' \notin \mathcal{F}_D\} \cup$$
$$\{(q, (\varsigma, \epsilon), q') \mid \exists q'' \in \mathcal{F}_D : (q'', \epsilon, q') \in \delta_D \wedge (q, \varsigma, q'') \in \delta_D\}$$

During the last step we replace the ϵ-transitions with transitions that directly approach the goals of the ϵ-transitions. This corresponds to the mode transition system semantics. A mode change is only possible after evaluating some event. With ϵ-transitions present, a spontaneous change via multiple modes would be possible.

EXAMPLE 6.10 (REMOVING THE HISTORY VARIABLE (CONT.))
Finally we remove ϵ-transitions and assign the mapping between modes and their behaviors.

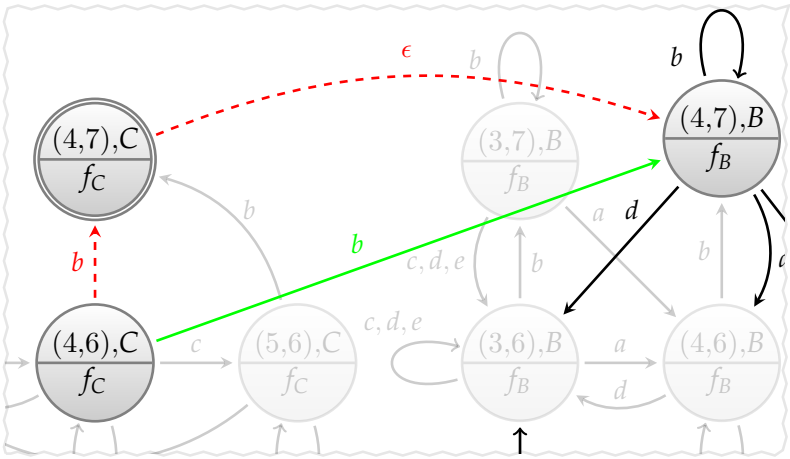


Figure 6.11.: Transfering the intermediate automaton into the final mts without a history variable

---

Isolated modes can exist as a result of the product automaton construction because the transition labels of the integrated automata are contradictory. Those modes can be removed if they are not initial i.e. they are not the result of combining the initial states of all transition automata (for an example of such initial states see states $((3,6),B)$ and $((3,6),C)$ in Figure 6.10).

THEOREM 6 (CORRESPONDENCE OF MODE TRANSITION SYSTEMS)
We proof the appropriateness of the transformation. Let $S = (\mathbb{M}, \delta, \Phi)$ be a mode transition system with pursue flags. After the transformation the resulting mode transition system $S' = (\mathbb{M}', \delta', \Phi')$ corresponds to the original mode transition system with respect to the activation of the mode-behaviors.

Again we require a relation $\mathcal{R} \in (\mathbb{M} \times \mathbb{H}(C_S)) \times \mathbb{M}'$ that relates mode and history variable of the original mode transition system to states of the transferred mode transition system without the history variable

$$
\begin{aligned}
&((m,h),n) \in \mathcal{R} \Rightarrow \Phi(m) = \Phi'(n) \wedge \\
&\qquad \forall e \in \Sigma, (m',h') \in succ_S((m,h),e)\, \exists (n', \langle\!\langle\rangle\!\rangle) \in succ_{S'}((n, \langle\!\langle\rangle\!\rangle), e) : \\
&\qquad ((m',h'),n') \in \mathcal{R}
\end{aligned}
$$

and regard the transformed mode transition system as appropriate iff:

$$\forall\, m \in \mathbb{M}, h \in \Sigma^* \,\exists\, n \in \mathbb{M}' : ((m,h),n) \in \mathcal{R}$$

PROOF 6.1:
We proof the property by induction over the history variable h. Let $D = (\mathcal{Q}_D, \Sigma, \delta_D, q_{0_D}, \mathcal{F}_D)$ *be the automaton modeling the transitions as described above. We define the relation* $\mathcal{R}$ *as follows.*

$$\begin{aligned}\mathcal{R} \stackrel{def}{=} &\{((m,\langle\!\langle\rangle\!\rangle),(q_i,m)) \mid m \in \mathbb{M} \wedge (q_i,m) \in q_{0_D}\} \cup \\ &\{((m,h),(q_i,n)) \mid m \in \mathbb{M} \wedge \exists (q_0,n_0) \in q_{0_D} : ((q_0,n_0),h,(q_i,n)) \in \delta'^*\}\end{aligned}$$

To keep the proof readable de define

$$q_i = (q_{1_i}, \ldots, q_{m_i}, \ldots, q_{|\mathbb{M}|_i})$$

which is a tuple combining states of all automata for all modes $m \in \mathbb{M}$. *By* $q_{i\lfloor m} = q_{m_i}$ *we denote the projection of a state* q_i *to the part of it that starts from* F_m *according to the product automaton construction.*

BASE CASE: *For the empty history variable (*$h = \langle\!\langle\rangle\!\rangle$*) obviously we find*

$$\{((m,\langle\!\langle\rangle\!\rangle),(q_0,m)) \mid m \in \mathbb{M} \wedge (q_0,m) \in q_{0_D}\} \in \mathcal{R}$$

Furthermore, we defined

$$\Phi'((q,m)) = \Phi(m)$$

Therefore, the assigned mode-behaviors match for the two states.

INDUCTION HYPOTHESIS: *For any h with* $\#(h) = i$*:*

$$\begin{aligned}((m,h),n) \in \mathcal{R} \Rightarrow \Phi(m) = \Phi'(n) \wedge \\ \forall\, e \in \Sigma, (m',h') \in succ_S((m,h),e)\, \exists (n',\langle\!\langle\rangle\!\rangle) \in succ_{S'}((n,\langle\!\langle\rangle\!\rangle),e) : \\ ((m',h'),n') \in \mathcal{R}\end{aligned}$$

INDUCTION STEP: *we proof the extension of h to h′ with length* $\#(h') = i+1$ *i.e. we regard a* $h' = h \ll e$*. By the induction hypothesis we know that* $((m',h'),n') \in \mathcal{R}$*. What we need to proof is that also*

$$\begin{aligned}&\Phi(m') = \Phi'(n') \wedge \\ &\forall\, e \in \Sigma, (m'',h'') \in succ_S((m',h'),e)\, \exists (n'',\langle\!\langle\rangle\!\rangle) \in succ_{S'}((n',\langle\!\langle\rangle\!\rangle),e) : \\ &((m'',h''),n'') \in \mathcal{R}\end{aligned}$$

We regard two cases:

Case 1: Firs we investigate the case where the input (here h') can still be extended to satisfy some mode transition condition:

$$\exists (m', (c_i, \rho), m'') \in \delta : \exists\, w \in L_{c_i} : h' \sqsubset w$$

We identify three sub-cases:

Case 1.1: $\exists\, w \in L_{c_i} : h'' = w$*, i.e., the extension of h' by e directly satisfies a transition condition. In that case:*

$$succ_S((m', h'), e) = \{(m'', h' \gg e) \mid \exists (m', (c_i, \rho), m'') \in \delta : h' \gg e \in L_{c_i}\}$$

Let $n' = (q_i, m')$. We know that

$$\forall\, r \in \mathbb{M} : (q_0 \lfloor_r, h', q_i \lfloor_r) \in \delta_r^*$$

and that there exists some

$$(q_i \lfloor_r, e, q_j \lfloor_r) \in \delta_r$$

For the mode under consideration this is especially [5]:

$$(q_i \lfloor_{m'}, e, q_j \lfloor_{m'}) \in \delta_{m'} \text{ with } q_j \lfloor_{m'} \in \mathcal{F}_{m'}$$

Hence, by the transitivity of any δ_r we know that

$$(q_0 \lfloor_r, h'', q_j \lfloor_r) \in \delta_r^*$$

By the definition of $\mathcal{M}_{m'}$ and $\mathcal{F}_D$ any state (q_j, m') with $q_j \lfloor_{m'} \in \mathcal{F}_{m'}$ is labelled as a final state in $\mathcal{F}_D$ and a ϵ-transition $(q_j, m') \to (q_j, m'')$ exists in δ_D. Due to skipping final states in the transferred mode transition system and directly proceeding to the destination of the ϵ-transition:

$$succ_{S'}((n', \langle\!\rangle), e) = ((q_j, m''), \langle\!\rangle) \text{ and}$$

$$((m'', h''), (q_j, m'') \in \mathcal{R}$$

i.e., all states that correspond to some transition automaton have proceeded according to the input and the mode has changed.

By the definition of Φ' we know that $\Phi'(q_j, m'') = \Phi(m'')$

Case 1.2: $\exists\, w \in L_{c_i} : h'' \sqsubset w$ *i.e., the extension by e does not yet satisfy any transition condition but might do so in the future. In that case:*

$$succ_S((m', h'), e) = \{(m', h' \gg e)\}.$$

Let $n' = (q_i, m')$. We know that

$$\forall\, r \in \mathbb{M} : (q_0 \lfloor_r, h', q_i \lfloor_r) \in \delta_r^*$$

and that there exists some

$$(q_i \lfloor_r, e, q_j \lfloor_r) \in \delta_r$$

[5] Note, for the proof the exact values of $q_i \lfloor_r$ and $q_j \lfloor_r$ or $q_i \lfloor_{m'}$ and $q_j \lfloor_{m'}$ are irrelevant. It is only important that there exists a valid value. For the proof it does not matter if we investigate the sequence of states of the D_m at the same time or one after another as soon as their respective mode becomes active

For the mode under consideration this is especially [5]*:*

$(q_i\lfloor_{m'}, e, q_j\lfloor_{m'}) \in \delta_{m'}$ *with* $q_j\lfloor_{m'} \in (\mathcal{Q}_{m'} \setminus \perp_{m'} \setminus \mathcal{F}_{m'})$

Hence, by the transitivity of δ_r *we know that*

$(q_0\lfloor_r, h'', q_j\lfloor_r) \in \delta_m^*$

Since no transition condition is satisfied in the original transition automaton $F_{m'}$ *will not reach a final state. Hence, and according to the definition of* $\mathcal{M}_{m'}$ *the next state* (q_j, m') *of* D_{TR} *is not flagged as a final state and no* ϵ*-transition exists to some state* (q_j, m'') *for any* $m'' \in \mathbb{M}$*.*

Hence,

$succ_{S'}(((q_i, n'), \langle\!\langle\rangle\!\rangle), e) = ((q_j, m'), \langle\!\langle\rangle\!\rangle)$ *and*

$((m', h''), (q_j, m') \in \mathcal{R}$

By the definition of Φ' *we know that* $\Phi'(q_j, m') = \Phi(m')$

Case 1.3: $\nexists\ w \in L_{c_m} : h' \sqsubset w$ *i.e. after the extension by e* h' *will never satisfy any transition condition in the future*

$succ_S((m', h'), e) = \{(m', h' \gg e)\}.$

Let $n' = (q_i, m')$*. We know that*

$\forall\, r \in \mathbb{M} : (q_0\lfloor_r, h', q_i\lfloor_r) \in \delta_r^*$

and that there exists some

$(q_i\lfloor_r, e, q_j\lfloor_r) \in \delta_r$

For the mode under consideration this is especially [5]*:*

$(q_i\lfloor_{m'}, e, \perp_{m'}) \in \delta_{m'}$

Hence, by the transitivity of δ_r *we know that*

$(q_0\lfloor_r, h'', q_j\lfloor_r) \in \delta_m^*$

Since no transition condition is satisfied in the original transition automaton $F_{m'}$ *will not reach a final state. Hence, and according to the definition of* $\mathcal{M}_{m'}$ *the next state* (q_j, m') *of* D_{TR} *is not labelled as a final state and no* ϵ*-transition exists to some state* q_j, m'' *for any* $m'' \in \mathbb{M}$*.*

Hence,

$succ_{S'}(((q_i, n'), \langle\!\langle\rangle\!\rangle), e) = ((q_j, m'), \langle\!\langle\rangle\!\rangle)$ *and*

$((m', h''), (q_j, m') \in \mathcal{R}$

By the definition of Φ' *we know that* $\Phi'(q_j, m') = \Phi(m')$

Case 2: Second we investigate the case $\nexists\ w \in L_{c_m} : h \sqsubset w$*. Here we know that the mode transition system will stay in the mode for ever because none of the outgoing transition labels will ever be satisfied.*

$succ_S((m', h'), e) = \{(m', h' \gg e)\}.$

Let $n' = (q_i, m')$. We know that

$$q_i|_{\underline{m'}} = \perp_{m'},$$
$$\forall\, r \in \mathbb{M} : (q_0|_{\underline{r}}, h', q_i|_{\underline{r}}) \in \delta_r^*$$

and that there exists

$$(q_i|_{\underline{r}}, e, q_j|_{\underline{r}}) \in \delta_r$$

For the mode under consideration this is especially [5]:

$$(\perp_{m'}, e, \perp_{m'}) \in \delta_{m'}$$

Hence, by the transitivity of any δ_r we know that

$$(q_0|_{\underline{r}}, h'', q_j|_{\underline{r}}) \in \delta_m^*$$

Furthermore, by the definition of $\mathcal{M}_{m'}$ and $\mathcal{F}_D$ we know that q_j is not labelled as a final state. In fact we can derive that no state that is reachable from q_j is labelled as a final state i.e. there is no way to reach a ϵ-transition such that some (q_k, m'') with $m'' \neq m'$ becomes an active state. This meets our expectations that no more mode changes are possible. Hence,

$$succ_{S'}(((q_i, n'), \langle\!\rangle), e) = ((q_j, m'), \langle\!\rangle) \text{ and}$$

$$((m', h''), (q_j, m')) \in \mathcal{R}$$

By the definition of Φ' we know that $\Phi'(q_j, m') = \Phi(m')$ *q.e.d.*

6.2.4. Executing analyses of the adaptation-subsystem

Now that we prepared the mode transition systems, i.e., we removed the bulky history variable, we are able to execute different kinds of analyses.

6.2.4.1. Consistency checking of (hierarchic) mode transition systems

First we check the consistency of mode transition systems. The consistency of mode transition systems is violated if two mode transition systems use the same history variable and mode pointer but require different transitions. This easily happens in a parallel composition that combines different views on the same modes.

Two mode transition systems with shared mts-variables are free of conflicts if their possible transitions coincide. This is true if either

- the mode transition systems allow the same transitions or
- non-determinism is reduced

Mode transition systems are inherently input enabled because they stay in a mode until contextual inputs satisfy some outgoing transition condition i.e. they are able to continue for every contextual input possibly by staying in the current mode for ever. We use this property: combining any two mode transition systems must not result in a partial transition behavior.

If two mode transition systems use a common mode pointer, it may happen that one mode transition system requires a mode change while the other one stays in the mode (or changes to a different mode). Hence, no valid valuation for the mode pointer exists. This is an inconsistency and breaks the input enabledness property of mode transition systems.

To investigate the consistency of mode transition systems we use the mode normal form and abstract from the actual terminal mode behaviors. Using the MNF has a great benefit. In general we need to mutually check any two mode transition systems with common mode-variables that may be active at the same time. In MNF this reduces to checking mode transition systems on paths from the root to a leaf with shared mode-variables. Mode transition systems existing in different branches never are active at the same time and never conflict. Furthermore, any two mode transition systems on a path without common mts-variables are without conflict.

Let $M = (\mathbb{M}_M, \delta_M, \Phi_m)$ and $N = (\mathbb{M}_N, \delta_N, \Phi_N)$ be two mode transition systems using the same modepointer PC and history variable h. Furthermore let $S = (G, \Psi)$ be a service hierarchy in mode normal form where $G = (V, A, \phi)$.

If $M, N \in V$ and $(M, N) \in A^*$ or $(N, M) \in A^*$ we create a set of mappings

$$MODES \subseteq \mathbb{M} \times \mathcal{P}(\Sigma^*) \to \mathcal{P}(\mathbb{M})$$

between languages and successor modes for a mode $m \in \mathbb{M}_M$ as follows:

Initially:

$$\forall (m, (c, \varsigma), m') \in \delta_M : MODES_m(L_c) = \{m'\}$$

We iteratively replace any two elements L_{c_i} and L_{c_j} within the domain of $MODES_m$ and their mappings as follows:

$$MODES_m(L_{c_i} \cap L_{c_j}) \mapsto MODES_m(L_{c_i}) \cup MODES_m(L_{c_j})$$
$$MODES_m(L_{c_i} \backslash L_{c_j}) \mapsto MODES_m(L_{c_i})$$
$$MODES_m(L_{c_j} \backslash L_{c_i}) \mapsto MODES_m(L_{c_j})$$

and repeat this until all the elements of the domain are disjoint. For N we execute the same steps. By this we handle possible overlapping languages of transition labels. The result is a function that maps a unique set of successors to a set of input words.

We extend the mapping by:

$MODES_m(\overline{L_m}) = \{m\}$
With $\overline{L_m} = \Sigma^* \setminus \bigcup L_{c_i}$

to capture inputs where the mode transition systems stay in their modes.

DEFINITION 42 (CONSISTENCY OF MODE TRANSITIONS):
Let $M = (\mathbb{M}, \delta_M, \Phi_M)$ and $N = (\mathbb{M}, \delta_N, \Phi_N)$ be two mode transition systems. The transitions of M and N are consistent if for all $m \in \mathbb{M}$ and $n \in \mathbb{N}$:

$$\forall L_i \in dom(MODES_m), L_j \in dom(MODES_n) :$$
$$L_i \cap L_j \neq \varnothing \Rightarrow MODES_m(L_i) \cap MODES_n(L_j) \neq \varnothing \qquad \wedge$$
$$\forall m \in \mathbb{M}, n \in \mathbb{N} : m = n \Rightarrow \Phi_M(m) = \Phi_N(n) \qquad \square$$

EXAMPLE 6.11 (CONSISTENCY OF MODE TRANSITION SYSTEMS)
As an example we use the clipping of two mode transition systems M and N from Figure 6.12. They use the same mode-variables.

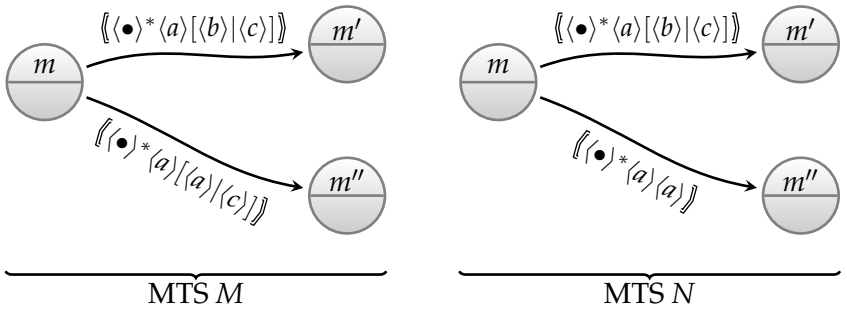


Figure 6.12.: Two clippings of mode transition systems using the same mode-variables

Mode transition system M has a non-determinism that is not present in mode transition system N. After handling overlapping languages we find:

Mode Transition System M	*Mode Transition System N*
$\llbracket\langle\bullet\rangle^*\langle a\rangle\langle b\rangle\rrbracket \mapsto \{m'\}$	$\llbracket\langle\bullet\rangle^*\langle a\rangle[\langle b\rangle\|\langle c\rangle]\rrbracket \mapsto \{m'\}$
$\llbracket\langle\bullet\rangle^*\langle a\rangle\langle c\rangle\rrbracket \mapsto \{m', m''\}$	$\llbracket\langle\bullet\rangle^*\langle a\rangle\langle a\rangle\rrbracket \mapsto \{m''\}$
$\llbracket\langle\bullet\rangle^*\langle a\rangle\langle a\rangle\rrbracket \mapsto \{m''\}$	

For brevity we omit staying in modes in the example and see that for any of the transitions the mode transition systems agree for a successor mode. ♣

We can execute this analysis already before the preparations of the mode transition system as described in Section 6.2.3. However, after applying the transformations, the analysis is easier. Let $M = (\mathbb{M}, \delta_M, \Phi_M)$ and $N = (\mathbb{M}, \delta_N, \Phi_N)$ be two mode transition systems that use common mode transition system variables. Let further be $M' = (\mathbb{M}', \delta'_M, \Phi'_M)$ and $N' = (\mathbb{N}', \delta'_N, \Phi'_N)$ be the mode transition systems after applying the transformations.

Starting in their initial modes, it must never happen that the two mode transition systems reach states that disagree in the part of their mode variable that encodes the original mode before the transformation. To check this we define a relation $\mathcal{R}$ as follows:

$$
\begin{aligned}
&\mathcal{R} \subseteq \mathbb{M}' \times \mathbb{N}' \\
&((p,q)),(r,s)) \in \mathcal{R} \Leftrightarrow \\
&\quad q = s \wedge \\
&\quad \forall\, c \in \Sigma\ \exists((p,q),(c,\rho),(p',q')) \in \delta'_M, ((r,s),(c,\rho),(r',s')) \in \delta'_N : \\
&\qquad ((p',q'),(r',s')) \in \mathcal{R}
\end{aligned}
$$

Now we investigate the two mode transition systems:

$$
\begin{aligned}
&\forall (p,q) \in \mathbb{M}', (r,s) \in \mathbb{N}' : \\
&p = q = r = s \Leftrightarrow ((p,q)(r,s)) \in \mathcal{R}
\end{aligned}
$$

Starting in the states that correspond to the original modes, this check ensures for all reachable states of the two product automata the equal mapping to modes according to the original mode transition systems. If this is true, the modes correspond in the original mode transition systems for each context information. More precisely, the value that both mode transition systems require for the common mode variables is equal.

Note that the first part of the modes in the transformed mode transition systems (here p' and r') in general need not to coincide. They are created by the intermediate states during the transformations and may use different names. Only the starting points of the check $((p,q)(r,s)) \in \mathbb{M}_A$ satisfy $p = q = r = s$. Those states represent the original initial states in the original mode transition systems where the history variable is empty.

We create two mode transition systems M_X with $X \in M, N$ that contain only the commonly accepted transitions but maintain the original mapping of the mode behaviors as follows:

$$
\begin{aligned}
&\mathbb{M}_X = \mathbb{M} \times \mathbb{N} \\
&\delta_X = \{(p,q),(r,s)),c,(p',q'),(r',s')) \mid \\
&\qquad ((p,q),c,(p',q')) \in \delta_M \wedge ((r,s),c,(r',s')) \in \delta_N \wedge q' = s'\} \\
&\Phi_M = \{((m,n),S) \mid S = \Phi_M(m)\} \\
&\Phi_N = \{((m,n),S) \mid S = \Phi_N(n)\}
\end{aligned}
$$

We replace the original two mode transition systems in the service hierarchy by the new mode transition systems M_M and M_N respectively. This harmonizes the sets $\mathbb{M}_M$ and $\mathbb{M}_N$. Of course we want to ensure equal valuations for the common mode variables even after the transformation.

Note that we do not join them. In the service hierarchy there might exist other mode transition systems between the two mode transition systems. Joining M_M and M_N requires the correct consideration of the mode transition systems in-between. This is possible but needs additional efforts without a major benefit for the further analysis.

6.2.4.2. Activation correctness

With the transformation of the mode transition systems, analyzing the activation correctness becomes more feasible. Without the history variable, the state space is finite. Yet the mode transition systems are still in a hierarchy. We need to flatten the hierarchy to make the resulting mode transition systems accessible to, e.g., model checkers.

We assume that the system specification is already in MNF and the transformations for removing the history variable were executed. Furthermore, we assume that the check for consistency of mode transition systems was executed successfully i.e. all mode transition systems in the hierarchy are consistent and the mode transition systems with common mode transition system variables were replaced as suggested.

In a hierarchy of mode transition systems the active modes at each level of the hierarchy determine the active services of the core system i.e. if services corresponding to entries in the contextual requirements chunks are active or not. Figure 6.13 illustrates the situation. The red framed nodes determine the active services in the core system.

EXAMPLE 6.12 (FLATTENING A HIERARCHIC MODE TRANSITION SYSTEM)
As an example we investigate a simple mode transition system without direct relation to the case study. Figure 6.13 presents a hierarchic mode transition system with two levels and core system services associated with the terminal modes. The red line indicates the active core system service that is determined by the active mode at the first level and the active mode at the second level. ♣

To flatten the hierarchic mode transition systems we apply yet another transformation which is based on product automata. Let $((V, A, \phi), \Psi)$ be the service hierarchy in MNF with root $r \in V$ and all terminal mode behaviors removed.

For the upcoming transformation, separate mode transition systems must have disjoint sets of modes i.e.

$$\forall i, j \in V : \mathbb{M}_i \cap \mathbb{M}_j \neq \varnothing \Rightarrow PC_i = PC_j$$

We allow that mode transition systems that act on the same mode transition system variables have common sets of modes of course.

Each $v \in V$ relates to an according mode transition system i.e., $\Psi(v) = (\mathbb{M}_v, \delta_v, \Phi_v)$.

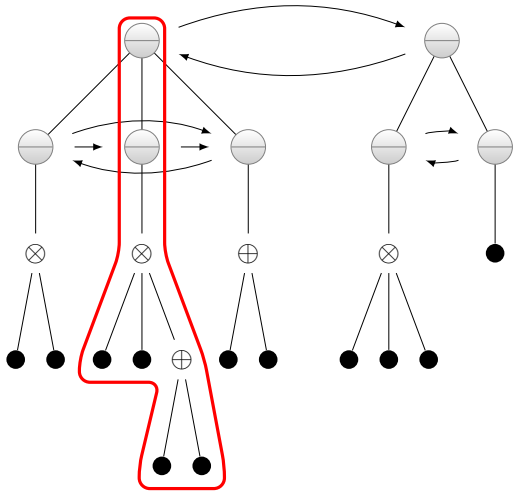


Figure 6.13.: The active terminal mode determines the active services of the core system

Let $MODES = \bigcup_{v \in V} \mathbb{M}_v$ be the set of all modes that appear within mode transition systems in the service hierarchy. We define an auxiliary relation

$$
\begin{aligned}
&\mathit{active\_modes} \in MODES \times \mathcal{P}(MODES)\\
&\mathit{active\_modes}(m) \mapsto Y \text{ such that}\\
&Y = \mu Y.\{m\} \cup\\
&\qquad \{n \in MODES \mid\\
&\qquad \exists_1 o \in Y, v_1, v_2 \in V : o \in \mathbb{M}_{v_1} \wedge v_2 = \Phi_v(o) \wedge n \in \mathbb{M}_{v_2}\}
\end{aligned}
$$

that takes a mode m as an argument and returns a set of modes that *possibly* can be active at the same time in a hierarchy of mode transition systems. This is determined by following down branches in a hierarchy of modes that is co-defined by the hierarchy of services. Note that *active_modes* is set valued. Each inner node has possibly multiple children each qualifying for a different result of *active_modes*. This includes the modes of the root mode transition systems.

Based on this auxiliary function we define a set $M_{active} \subset \mathcal{P}(MODES)$ that gathers sets of modes that can be active at the same time. Formally:

$$M_{active} = \{\mathit{active\_modes}(m) \mid m \in \mathbb{M}_r\}$$

EXAMPLE 6.13 (FLATTENING A HIERARCHIC MODE TRANSITION SYSTEM CONT.)
To illustrate the function active_modes and the set M_{active} we extend the hierarchy by one level. Figure 6.14 illustrates three results of the function active_modes, two results for the left mode of the root mode transition system and one result for the right mode. All three are elements of M_{active} ♣

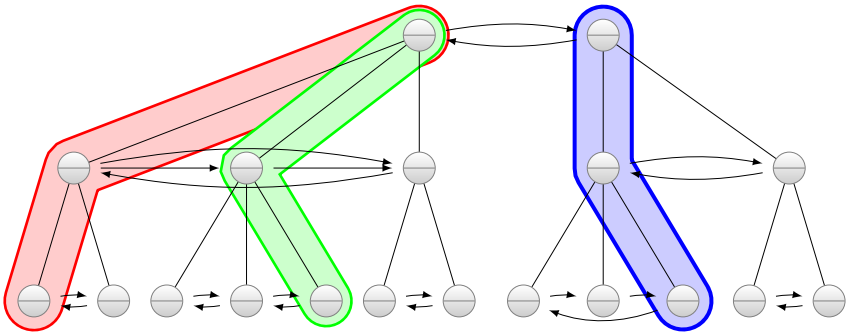


Figure 6.14.: Three active mode sets, two (red and green) starting in the same mode of the root service and one starting in a different one (blue)

Let $(\mathcal{S}_i)_{i\in I}$ be a set of sets with $I \subset \mathbb{N}$ as an index to these sets. We define an auxiliary function:

$$\prod\!\!\!\!\bullet : (\mathcal{S}_i)_{i\in I} \to \mathcal{P}(\bigcup_{i\in I} S_i),$$

$$\prod_{i\in I}\!\!\!\!\bullet\, S \stackrel{def}{=} \{\{m_1, \ldots m_n\} \mid \forall\, i \in [1,n], S_i \in S\, \exists_1\, m_i : m_i \in S_i\}$$

We call this operation the *element-wise union*. The idea is that the element-wise union creates sets that contain exactly one element of the sets that appear in $\mathcal{S}$. The operation is similar to the Cartesian product but returns sets rather than tuples such that the order of the elements is irrelevant.

Using this auxiliary function we define a mode transition system $F = \mathbb{M}_F, \delta_F, \Phi_F$ as follows:

$$\mathbb{M}_F = \prod_{i\in|\mathbb{M}|}\!\!\!\!\bullet\, \mathbb{M} \text{ where } \mathbb{M} = \bigcup_{v\in V}\{M_v\}$$

$$\delta_F = \{(p,c,q) \mid \exists\, modes, modes' \in M_{active} : modes \subseteq p \wedge modes' \subseteq q \quad \wedge$$
$$\forall\, m \in modes\, \exists\, v \in V : (m,c,m') \in \delta_v \wedge m' \in q\}$$
$$\Phi_F = \{(q,S) \mid \exists\, modes \in M_{active}, v \in V, m \in modes :$$
$$modes \subseteq q \wedge leave(v) \wedge m \in \mathbb{M}_v \wedge S = \Phi(m)\}$$

The transformed mode transition system executes transitions such that all modes of the active mode transition systems in the hierarchy switch to their correct successor. All mode transition systems that are not active may transit to an arbitrary mode. This matches to the definition of services where uncontrolled variables take arbitrary values. The mode behaviors in the transformed mode transition system correspond to the terminal mode behaviors.

EXAMPLE 6.14 (FLATTENING A HIERARCHIC MODE TRANSITION SYSTEM CONT.)
We demonstrate the corresponding transitions in the original hierarchy. Figure 6.15 shows the three transitions in the original hierarchy that define the corresponding transition in the transformed mode transition system. The flattened mode transition system combines modes

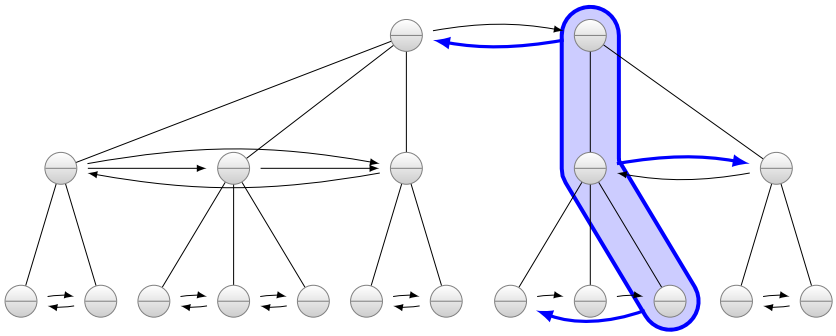


Figure 6.15.: Transitions of the original hierarchy influencing the transitions of the flattened mts

of all original mode transition systems into its modes. The blue framed modes represent one possible combination of active modes. ♣

After the transformation, e.g., a model checker can be extended to provide an answer to the question if it is always true if a contexts entry condition is received that some mode is active, whose mapping in Φ has a mode behavior that contains the service that belongs to the context. Furthermore, it can be checked if a contexts exit condition always leads to the deactivation of the corresponding service.

6.3. Analyzing the core system

In this section we change from analyzing the adaptation-subsystem to analyzing the core system. We base the analysis on the MNF as well. On benefit of this is that the terminal mode behaviors are separated and cannot interfere. This allows focusing the analysis on smaller parts.

6.3.1. Consistency

We start with the analysis for consistency. Before developing a preliminary specification into a complete one we need to reveal all contradictions. Otherwise no proper implementation exists and applying chaos completion hides conflicting requirements. Informally, two or more requirements simultaneously imposing different demands to the system cause a conflict. In the context of MARLIN with services formalizing requirements any two services controlling the values of shared resources like variables

and channels and defining incompatible values cause conflicts. Values are incompatible if they are disjoint. Similar to [Schätz and Salzmann, 2003] we define inconsistency as follows.

DEFINITION 43 (INCONSISTENCY/CONFLICT):
Without restriction, let there be two services $A \in \mathbb{F}(I_A \rhd O_A, V_A)$ and $B \in \mathbb{F}(I_B \rhd O_B, V_B)$. We say that A and B are inconsistent (or conflicting) with respect to their parallel composition iff:

$$\exists(\sigma, x) : (\sigma_{\lfloor V_A}, x_{\lfloor I_A}) \in dom(A) \vee (\sigma_{\lfloor V_B}, x_{\lfloor I_B}) \in dom(B)) \wedge (\sigma, x) \notin dom(A \otimes B)$$

*we write $A \not\rightleftharpoons B$ and say that the composition of A and B introduces **additional** partiality.* □

If only parallel composition was available in MARLIN this definition would already imply a methodological approach for revealing conflicts. By pairwise checking all services for inconsistencies we could either show that the complete specification is consistent or we could allocate the inconsistency. However, MARLIN allows to supersede known conflicts by an alternative composition of additional services (e.g., as shown in example 5.4). Therefore, we need a careful analysis of the specification.

DEFINITION 44 (AGGREGATED INCONSISTENCY):
A service S that is built up from sub-services $S_1 \ldots S_n$ has an aggregated inconsistency iff

$$\exists\, S_i \in \{S_1, \ldots, S_n\}, (\sigma, x) \in \Sigma_S \times \vec{I_S} :$$
$$(\sigma_{\lfloor V_i}, x_{\lfloor I_i}) \in dom(S_i) \wedge (\sigma, x) \notin dom(S)$$

Informally spoken, there is a service S_i whose domain contains at least one element that is not in the domain of the compound service S. We write $S_i \not\prec S$ □

The existence of services that override inconsistencies by alternative composition requires the identification of all sub-services that are candidates for inconsistencies and all possible services that override the conflicts.

The transformation of a specification into MNF allows the separate analysis of behaviors in different modes. No two terminal mode behaviors of a specification in MNF are active at the same time. Services that are never active together cannot conflict. Since the identification of causes for conflicts requires a mutual analysis of pairs of services reducing the set of services to be analyzed considerably reduces the efforts for the analysis.

Still a specification of the behavior in a terminal mode can be complex and the mix of alternative and parallel composition complicates the analysis. Therefore, we introduce another normal form that is applicable to terminal mode behaviors and organizes services in a useful way.

DEFINITION 45 (PARALLEL NORMAL FORM):
A service is in parallel normal form (PNF) if it is free from modes and is a parallel composition of alternative compositions of atomic services.

- *Any atomic service AS is in parallel normal form*
- *A compound service is in parallel normal form if it is of the form*

$$(\otimes_{i=1}^{n}(\oplus_{j=1}^{m_i}F_{i,j}))$$

Where all $F_{i,j}$ are atomic services □

THEOREM 7
For every service S that is free from modes exists an equivalent service S′ that is in parallel normal form.

PROOF 7.1 (OF THEOREM 7):
Proof by induction on the structure of the formulas. The proof-schema is similar to the proof of the existence of the disjunctive normal form for formulas in propositional logic (see, e.g., [Schoning, 2000])

Base case: By definition, any atomic service already is in PNF.

Inductive step: We distinguish two cases:

Case 1: The service S is of the form $A \oplus B$.
By induction hypothesis there exist

$$A' = (\otimes_{i=1}^{n}(\oplus_{j=1}^{m_i}F_{i,j})) \text{ and } B' = (\otimes_{k=1}^{o}(\oplus_{l=1}^{p_k}F_{k,l}))$$

that are equivalent to A and B and are in PNF. We substitute:

$$S = (\otimes_{i=1}^{n}(\oplus_{j=1}^{m_i}F_{i,j})) \oplus (\otimes_{k=1}^{o}(\oplus_{l=1}^{p_k}F_{k,l}))$$

and by repeated application of Theorem 15 we get

$$S = (\otimes_{i=1}^{n}(\otimes_{k=1}^{o}((\oplus_{j=1}^{m_i}F_{i,j}) \oplus (\oplus_{l=1}^{p_k}F_{k,l}))))$$

This service is in PNF and by application of the associativity law (cf. Theorem 14) can be restructured to

$$S = (\otimes_{i=1}^{n*o}(\oplus_{j=1}^{q_i}G_{i,j}))$$

with a proper renumbering of the atomic services.

Case 2: The service S is of the form $A \otimes B$.
Similar to case 1 by substitution we get

$$S = (\otimes_{i=1}^{n}(\oplus_{j=1}^{m_i} F_{i,j})) \otimes (\otimes_{k=1}^{o}(\oplus_{l=1}^{p_k} F_{k,l}))$$

which is in PNF and can be restructured to

$$S = (\otimes_{i=1}^{n+o+1}(\oplus_{j=1}^{q_i} G_{i,j}))$$

with a proper renumbering of atomic services. *q.e.d.*

All specifications can first be transformed into MNF. After that every terminal mode in the MNF is free from modes. Therefore, we may investigate every terminal mode behavior in PNF. *Each clause of the PNF contains all the options for the parallel services to react with respect to possible supplements.*

EXAMPLE 6.15 (PARALLEL NORMAL FORM)
Regard a service expression

$$(((A \otimes B) \oplus C) \otimes D) \otimes (((E \oplus F) \otimes G) \oplus H)$$

as it may appear in a terminal mode to motivate the use of the PNF. Assume that A and B are inconsistent but H supersedes the conflict. By applying the appropriate transformations we get the expression

$$(A \oplus C) \otimes (B \oplus C) \otimes D \otimes (E \oplus F \oplus H) \otimes (G \oplus H).$$ ♣

In complex expressions it is difficult to reveal conflicts. Therefore, we are interested in a sophisticated analysis method that is accessible to a structured approach and finally to tool support. Providing normal forms like the PNF aids this approach. This finding is heavily inspired by similar normal forms like clause normal form and conjunctive normal form being useful for answering questions in propositional and predicate logic [Harrison, 2009].

With the PNF we can mutually investigate each clause for conflicts to reveal them. The mutual investigation of clauses now is similar to the aforementioned situation with parallel composition only. Furthermore, we know that two services only may be conflicting if they share resources. Hence we can safely omit an investigation of clauses with different controlled resources.

EXAMPLE 6.16 (PARALLEL NORMAL FORM CONT.)
We assume that C does not contribute to the resolution of the conflict between A and B. Obviously H is inappropriate to solve the conflict between A and B completely because it finally is parallel composed and therefore only qualifies for further restrictions rather that superseding conflicts. ♣

DEFINITION 46 (FINDING AGGREGATED INCONSISTENCIES):
Let S be a specification in PNF:

$$S = (\otimes_{i=1}^{n}(\oplus_{j=1}^{m_i} F_{i,j}))$$

and $F_{i,j}$ an atomic services.
S has no conflicting sub-services iff

$$\forall k, l \in [1, n]:$$
$$\{(\sigma, x, y, \acute{\sigma}) \mid (\sigma, x, y, \acute{\sigma})_{\lfloor F_k} \in F_k\} =$$
$$\{(\sigma, x, y, \acute{\sigma}) \mid (\sigma, x, y, \acute{\sigma})_{\lfloor F_l} \in F_l\}$$

with $F_k = \oplus_{j=1}^{m_k} F_{k,j}$ and $F_l = \oplus_{j=1}^{m_l} F_{l,j}$ □

To this end we described a syntactical transformation that helps analyzing a specification for the existence of conflicts. To support this analysis, we describe a correlation on the semantic layer that operationalizes the analysis. The semantics of a service is a possibly infinite set of tuples $(\sigma, i, o, \acute{\sigma})$. Each tuple describes a behavior of arbitrary length. Any inconsistency may only manifest after some time. In general, we analyze all behavioral tuples to reveal conflicts. This includes an overhead of work. Due to the infix closure of the semantics the same conflict becomes obvious earlier for some other semantic tuples already.

We choose a big step semantics to take the peculiarities of contexts and the related concept of modes into account. Now that we are able to transform any specification into mode normal form the terminal modes' behaviors are free of this constraint. They only comprise alternative and parallel composition. In this setting we may switch to a small step semantics. We justify this switch by the theorem of propagation of inconsistencies.

THEOREM 8 (PROPAGATION OF INCONSISTENCIES)
For two services A and B it is sufficient to investigate the single steps of their behaviors

$$\{(\sigma_A, \langle x_A \rangle, \langle y_A \rangle, \acute{\sigma}_A) \mid (\sigma_A, \langle x_A \rangle, \langle y_A \rangle, \acute{\sigma}_A) \in [\![A]\!]\} \text{ and}$$
$$\{(\sigma_B, \langle x_B \rangle, \langle y_B \rangle, \acute{\sigma}_B) \mid (\sigma_B, \langle x_B \rangle, \langle y_B \rangle, \acute{\sigma}_B) \in [\![B]\!]\}$$

to find inconsistencies.

PROOF 8.1 (OF THEOREM 8):
An inconsistency has one of the following characteristics: I) the two services disagree on output channels or II) they disagree on final states or III) the input is not contained in the domain of one of the services.

Case 1: conflict on output channels

We regard all (σ, i) that are in the domain of both services, A, and B. If services A and B with shared output channels O disagree in the valuations of output channels then

$$\exists\, i, \sigma :\ \{o_{A\lfloor O} \mid (o_A, \acute{\sigma}_A) \in A.(\sigma_{\lfloor A}, i_{\lfloor A})\} \cap \{o_{B\lfloor O} \mid (o_B, \acute{\sigma}_B) \in B.(\sigma_{\lfloor B}, i_{\lfloor B})\} = \varnothing$$

By using the property of infix closeness we conclude:

$$\begin{aligned}\exists\, \sigma_n, t :\ & i_1 = (i)_{\downarrow t} \wedge i_2 = (i)^{\uparrow t} \wedge \\ & \{o_{A\lfloor O} \mid (o_A, \sigma_n) \in A.(\sigma_{\lfloor A}, i_{1\lfloor A})\} \cap \{o_{B\lfloor O} \mid (o_B, \sigma_n) \in B.(\sigma_{\lfloor B}, i_{1\lfloor B})\} \neq \varnothing \wedge \\ & \{o_{A\lfloor O} \mid (o_A, \acute{\sigma}_1) \in A.(\sigma_{n\lfloor A}, i_{2\lfloor A})\} \cap \{o_{B\lfloor O} \mid (o_B, \acute{\sigma}_B) \in B.(\sigma_{n\lfloor B}, i_{2\lfloor B})\} = \varnothing\end{aligned}$$

Hence t is the first time, where A and B disagree. Behaviors starting at t with a shared initial state are contained in both service. We may safely assume that the shared state σ_n exists since case 2 captures deviations. Again by infix closeness we know

$$\begin{aligned}\forall(\sigma, \langle i_1\rangle \circ i) :(\langle o_1\rangle \circ o, \acute{\sigma}) \in A.(\sigma, \langle i_1\rangle \circ i) \Rightarrow \exists\, \sigma_n : (\langle o_1\rangle, \sigma_n) \in A.(\sigma, \langle i_1\rangle) \wedge \\ (\langle o_2\rangle \circ o, \acute{\sigma}) \in B.(\sigma, \langle i_1\rangle \circ i) \Rightarrow \exists\, \sigma_n : (\langle o_1\rangle, \sigma_n) \in B.(\sigma, \langle i_1\rangle)\end{aligned}$$

If two services disagree on output channels after some time, we will always find a single step in the services' semantics that corresponds to the behavior in the time interval where the conflict happens first.

Case 2: conflict on final states.

The argument is similar to case 1. We regard all (σ, i) that are in the domain of both services. If two services A and B with shared variables V disagree on the valuations of some of the variables then

$$\exists\, i, \sigma :\ \{\acute{\sigma}_{A\lfloor V} \mid (o_A, \acute{\sigma}_A) \in A.(\sigma_{\lfloor A}, i_{\lfloor A})\} \cap \{\acute{\sigma}_{B\lfloor O} \mid (o_B, \acute{\sigma}_B) \in B.(\sigma_{\lfloor B}, i_{\lfloor B})\} = \varnothing$$

Again by using the infix closeness property we conclude.

$$\begin{aligned}\forall(\sigma, i \circ \langle i_1\rangle) :(o \circ \langle o_1\rangle) \in A.(\sigma, i \circ \langle i_1\rangle) \Rightarrow \exists\, \sigma_n : (\langle o_1\rangle, \sigma_n) \in A.(\sigma, \langle i_1\rangle) \wedge \\ (o \circ \langle o_2\rangle, \acute{\sigma}) \in B.(\sigma, i \circ \langle i_1\rangle) \Rightarrow \exists\, \sigma_n : (\langle o_1\rangle, \sigma_n) \in B.(\sigma, \langle i_1\rangle)\end{aligned}$$

Hence we find the same contradiction just by analyzing the single steps of the services.

Case 3: the input is out of the domain of one service. Without restriction we assume that the inputs are not in the domain of A.

$$\exists\, i, \sigma :\ (\sigma_{\lfloor V_B}, i_{\lfloor I_B}) \in dom(B) \wedge (\sigma_{\lfloor V_A}, i_{\lfloor I_A}) \notin dom(A)$$

This case has two sub-cases of which we just sketch both since they rely on the same arguments as for case 1 and case 2.

I) σ is responsible for $(\sigma, i) \notin dom(A)$. Only if A and B share variables this has an effect. Otherwise we always can find a σ whose projection to variables of A fits to some initial state of some behavior of B.

$$\exists(\sigma_B, i_B) \in dom(B) : \forall\, \sigma : \sigma_{\lfloor V_B} = \sigma_B \Rightarrow \nexists(\sigma_A, i_A) \in dom(A) : \sigma_{\lfloor V_A} = \sigma_A$$

i.e., there exists an initial state for B with no extension to variables of A such that a projection to the variables of A is a valid initial state for A. This immediately becomes apparent (right before the first transition) and hence it is sufficient to investigate all single step behaviors.

II) If i is responsible for $(\sigma, i) \notin dom(A)$ we apply the same argument as in case 1. We can find a prefix that is in the domain and an according state σ_n at the end of processing this prefix such that B is able to continue in σ_n but A is not (A is able to process the prefix of inputs but then lacks the capability to react at some time). By infix closure we may find behaviors in A starting at σ_n but none is capable of processing the rest of the input. Again by infix closure we can strip of the tail of the inputs. Hence, the single step behaviors are sufficient for the investigation. q.e.d.

We illustrate the analysis of services for consistency with an example from the case study.

EXAMPLE 6.17 (CONFLICT ANALYSIS)
To demonstrate the use of the parallel normal form we use the services for opening the front doors. For reasons of brevity we omit the full specification showing the interfaces and concentrate on the behavioral aspect. The two services `dd_open_vlowkey` *(cf. Figure 6.16(a)) and* `pd_open_vlowkey` *(cf. Figure 6.16(b))) control the opening of the driver door and the passenger door respectively. The two services are already compound:*

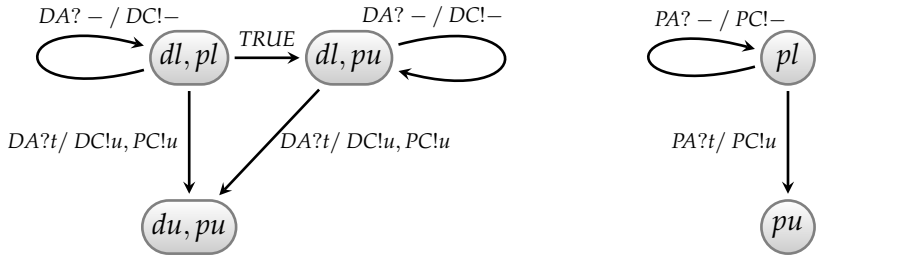
$$\texttt{dd\_open\_vlowkey} = \texttt{dd\_keepclose} \oplus \texttt{dd\_opentouch}$$
$$\texttt{pd\_open\_vlowkey} = \texttt{pd\_keepclose} \oplus \texttt{pd\_opentouch}$$

The service $\texttt{open\_front\_doors} = (\texttt{dd\_open\_vlowkey}) \otimes (\texttt{pd\_open\_vlowkey})$ *already is in parallel normal form. By analyzing the respective transitions we find that the two transitions*

1) $\{dd\_cl \times pd\_cl\}DA?t/CD!u, PC!u\{dd\_cu \times pd\_cu\}$
2) $\{pd\_cl\}PA? - /PC! - \{pd\_cl\}$

are in conflict. The conflict exists if someone touches the driver door but nobody touches the passenger door.

A similar observation is possible for the trunk door and the driver door but not for the passenger door and the trunk door.



(a) Opening the driver door if a key is present and the speed is low

(b) Opening the passenger door if a key is present and the speed is low

Figure 6.16.: Two aspects of opening the front doors with the potential for inconsistencies

We investigate a modified expression

$$\texttt{frt\_drs\_open\_vlowkey} = ((\texttt{dd\_keepclose} \oplus \texttt{dd\_opentouch}) \otimes (\texttt{pd\_keepclose} \oplus \texttt{pd\_opentouch})) \otimes (\texttt{DDPDFI})$$

with a service DDPDFI *like given in Example 5.4 overriding the conflict. The PNF reads:*

$$(\texttt{dd\_keepclose} \oplus \texttt{dd\_opentouch} \oplus \texttt{DDPDFI}) \otimes (\texttt{pd\_keepclose} \oplus \texttt{pd\_opentouch} \oplus \texttt{DDPDFI})$$

Comparing the domains of the services defined by each clause we find that the domains remain the same for a composition with respect to the projection to the respective interface. ♣

6.3.2. Input enabledness

The term "completeness" is ambiguous. A specification can be complete with respect to the requirements of any stakeholder without being input enabled. On the other hand, a specification can be input enabled but lacks some stakeholders' requirements. The initial set of requirements usually is incomplete with respect to both.

We already argued in Section 4.3 that we treat under-specification as partiality (closed world assumption). As a consequence the formal specification also is incomplete i.e. the function that defines the system's behavior is partial if the system is underspecified. Similar to conflicts, partiality prevents reactions to some inputs. In contrast to checking for input enabledness, there is no formal means to check for completeness with respect to the stakeholders' requirements.

From a methodological point of view partiality caused by underspecification is different from partiality caused by inconsistencies. Inconsistencies always indicate conflicting requirements. In contrast, incompleteness can be intentional if no stakeholder

has according requirements defined (yet). Causal chaos closure [Broy, 2005] assigns arbitrary behavior to these cases and completes a specification. We regard a specification as preliminary as long as it is partial.

Applying a closure before examining the underspecification is no good idea. Identified underspecification should be discussed with the stakeholders to clarify if the underspecification is intended. The analysis of the specification for partiality helps identifying those blind spots.

DEFINITION 47 (INPUT ENABLED):
Similar to [Schätz et al., 2003] we define a service S as input enabled iff

$$\forall(\sigma, x)\, \exists(y, \acute{\sigma}) : (\sigma, x, y, \acute{\sigma}) \in [\![S]\!]$$ □

The input enabledness of a specification only depends on the mode-behaviors because the mode transition systems are input enabled by construction – they remain in a mode with their mode behaviors active until an outgoing mode transition is enabled.

Again we investigate the terminal mode behaviors in MNF. We are interested in missing behaviors i.e. contexts where some inputs cannot be processed. Hence, comparing two services for restricting the compound domain – as we did for the consistency analysis – is inappropriate: even if the domain of the compound service fits to the domain of the operands, the service can be partial. Again we use a normal form for analyzing input enabledness.

DEFINITION 48 (ALTERNATIVE NORMAL FORM):
A service is in alternative normal form (ANF) if it is free from modes and is an alternative composition of parallel compositions of atomic services.

- *Any atomic service AS is in alternative normal form*
- *A compound service is in alternative normal form if it is of the form*

$$(\oplus_{i=1}^{n}(\otimes_{j=1}^{m_i} F_{i,j}))$$

Where all $F_{i,j}$ are atomic services □

THEOREM 9
For every service S that is free from modes there exists an equivalent service S′ is in alternative normal form.

PROOF 9.1 (OF THEOREM 9):
The proof is analogous to Proof 7.1 *q.e.d.*

The ANF organizes behaviors into alternative sets of behaviors. While the PNF organizes the specification of a mode behavior in a number of parallel automata, the ANF creates a single automaton with the transitions defined in the clauses. Conflicts between services within a clause remove the clause. Thus only effective clauses remain. In difference to the PNF the ANF does not allow an identification of conflicts. It may happen that some other clause exists that supersedes the conflict.

We use the effect of alternative composition to enrich behaviors with additional capabilities to react – both, as true alternatives but also in a sequential manner (cf. [Clarke et al., 1999]). The analysis searches for clauses that provide reactions to an input and state under consideration.

EXAMPLE 6.18 (ALTERNATIVE NORMAL FORM)
As a motivation, again consider the service expression

$$(((A \otimes B) \oplus C) \otimes D) \otimes (((E \oplus F) \otimes G) \oplus H)$$

by application of the transformation rule we end up with an expression in ANF like this:

$$(A \otimes B \otimes D \otimes E \otimes G) \oplus (A \otimes B \otimes D \otimes F \otimes G) \oplus$$
$$(A \otimes B \otimes D \otimes H) \oplus (C \otimes D \otimes E \otimes G) \oplus (C \otimes D \otimes F \otimes G) \oplus (C \otimes D \otimes H)$$

Each clause defines a possible empty behavior. The behaviors of the clauses create one large automaton that defines all available reactions. ♣

Similar to Theorem 8 in Section 6.3.1 (the infix closure of the services) we switch to the investigation of single steps.

THEOREM 10 (PROPAGATION OF INCOMPLETENESS)
A service S in ANF with the set of clauses CL is input enabled iff

$$\forall(\sigma, \langle i\rangle, \langle o\rangle, \acute{\sigma})\, \exists\, cl \in CL : (\sigma, \langle i\rangle, \langle o\rangle, \acute{\sigma}) \in [\![cl]\!]$$

The ANF contains each possible transition as a single step behavior in some clause.

PROOF 10.1 (OF THEOREM 10):
The proof is similar to Proof 8.1. Therefore we only sketch the idea.

We use the infix closure property to proof that valid input sequences can be split into a head and a tail with a valid intermediate state such that both, head and tail are valid behaviors as well. Therefore, a sequence of inputs is processable if a corresponding series of single steps exists that are valid behaviors. Hence, if an input sequence is not processable, an intermediate state exists such that the inputs up to this state are valid and the next input cannot be processed. q.e.d.

Again an appropriate proof system like Mona [Henriksen et al., 1995] can execute this analysis. Schätz, e.g., shows such an approach in [Schätz, 2009].

EXAMPLE 6.19 (COMPLETENESS ANALYSIS)
As an example we use the services for opening the front doors. Obviously they are partial. One easily can see this. However, we apply the structured approach to demonstrate the analysis.

Recall the two services `dd_open_vlowkey` *(cf. Figure 6.16(a)) and* `pd_open_vlowkey` *(cf. Figure 6.16(b))) control the opening of the driver door and the passenger door respectively. This time we assume the inconsistency (i.e., the feature interaction) is superseded by the service* `DDPDFI` *as presented in Section 5.2.1. The complete service expression reads:*

$$\begin{aligned}&\texttt{frt\_drs\_open\_vlowkey} = \\ &\quad ((\texttt{dd\_keepclose} \oplus \texttt{dd\_opentouch}) \otimes (\texttt{pd\_keepclose} \oplus \texttt{pd\_opentouch})) \otimes \\ &\quad (\texttt{DDPDFI})\end{aligned}$$

The according expression in ANF is:

$$\begin{aligned}&(\texttt{dd\_keepclose} \otimes \texttt{pd\_keepclose}) \oplus && (1)\\ &(\texttt{dd\_keepclose} \otimes \texttt{pd\_opentouch}) \oplus && (2)\\ &(\texttt{dd\_opentouch} \otimes \texttt{pd\_keepclose}) \oplus \\ &(\texttt{dd\_opentouch} \otimes \texttt{pd\_opentouch}) \oplus && (3)\\ &\texttt{DDPDFI} && (4)\end{aligned}$$

We drop the clause $(\texttt{dd\_opentouch} \otimes \texttt{pd\_keepclose})$ *because it is unsatisfiable. The service* `DDPDFU` *supersedes the conflict in another clause. Figure 6.17 shows the result of the transformation. We number transitions according to the clauses in the above expression. If we examine*

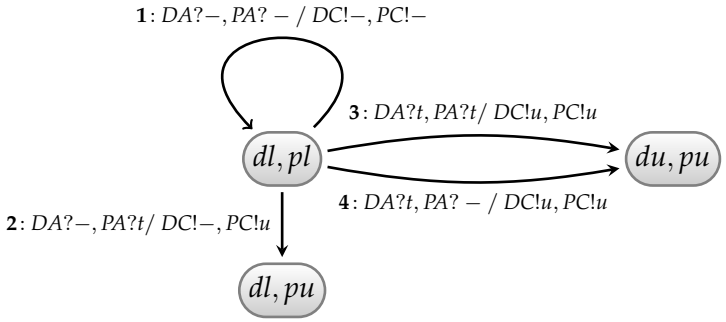


Figure 6.17.: Illustration of the behavior of the front doors after transforming the specification into ANF

the transitions starting in each state we find that the service is able to react to every possible input if in state `dl, pl` *i.e. for each input there exists a clause defining a behavior staring in this state and accepting the input. However, neither of the other states has outgoing transitions that allow a reaction to some input.* ♣

6.3.3. Small step semantics for the analysis of the system core

In the previous two sections we introduced a small step semantics for the analysis of conflicts and completeness. We strongly emphasize that we only proved the applicability for these two analyses. They may be applicable to analyzing other properties as well but we did not analyze this.

As an example where the small step semantics does not help regard the analysis of a minimal mode-behavior for continuing interactions until a mode change is triggered. If a service is input enabled it is capable of producing behaviors of arbitrary length and satisfies the property. However, it may happen that this is over-engineered. Assume the system executes a mode change as soon as the current mode-behavior emits a certain series of message. Furthermore, assume that all paths leading to a certain state emit these messages. In such a scenario it is unnecessary to require the state to be input enabled. If the state is changed in another mode, it is certain that the regarded state never is initial and therefore never needs outgoing transitions. To analyze this, we need to take into account the mode transition system together with all possible paths through the mode-behaviors.

We used the big step semantics to prove properties of the language, especially the existence and equivalence of normal forms. In practice it is easier to derive the single step semantics directly from the specification rather than utilizing the infix closure property and processing the big step semantics. If we are only interested in the conflicts and completeness it is useful to skip the derivation of the big step semantics and directly proceed with the small step semantics.

The set of single steps of a service easily can be derived by joining or intersecting the sets of single steps of atomic services according to their alternative or parallel composition. We do not need to involve the fixed point operation for deriving the small step semantics. We denote the small step semantics of a service S by $[\![S]\!]_{\langle\rangle}$ (compare to the definitions in Chapter 4):

$$
\begin{aligned}
[\![S]\!]_{\langle\rangle} \stackrel{def}{=} (\{(\sigma_S, \langle x_S\rangle, \langle y_S\rangle, \acute{\sigma}_S) \mid\ & \sigma_S \in \Sigma_S && \wedge\\
& x_S \in \mathbb{H}_2(I_S) && \wedge\\
& y_S \in \mathbb{H}_2(O_S) && \wedge\\
& \acute{\sigma}_S \in \Sigma_S && \wedge\\
& B && \}\,)
\end{aligned}
$$

Parallel and alternative composition become:

$$
\begin{aligned}
&[\![A \otimes B]\!]_{\langle\rangle} \stackrel{def}{=} \{(\sigma, x, y, \acute{\sigma}) \mid\\
&\qquad (\sigma_{\lfloor V_A}, x_{\lfloor I_A}, y_{\lfloor O_A}, \acute{\sigma}_{\lfloor V_A}) \in [\![A]\!]_{\langle\rangle} \wedge (\sigma_{\lfloor V_B}, x_{\lfloor I_B}, y_{\lfloor O_B}, \acute{\sigma}_{\lfloor V_B}) \in [\![B]\!]_{\langle\rangle}\}\\
&[\![A \oplus B]\!]_{\langle\rangle} \stackrel{def}{=} \{(\sigma, x, y, \acute{\sigma}) \mid\\
&\qquad (\sigma_{\lfloor V_A}, x_{\lfloor I_A}, y_{\lfloor O_A}, \acute{\sigma}_{\lfloor V_A}) \in [\![A]\!]_{\langle\rangle} \vee (\sigma_{\lfloor V_B}, x_{\lfloor I_B}, y_{\lfloor O_B}, \acute{\sigma}_{\lfloor V_B}) \in [\![B]\!]_{\langle\rangle}\}
\end{aligned}
$$

Without proof: we can derive the big step semantics for services that are FFM by applying a closure directly to the compound services of the terminal mode behaviors in MNF. It is not necessary to derive the big step semantics for the sub-services already. A proof is similar to the proof of Theorem 8.

The small step semantics bases on the semantics presented in [Schätz, 2009, 2007, 2006]. As a main difference to these approaches we do not insist in an explicit representation of control points. We encode them implicitly (if needed) as state variables.

The single step semantics ease the analysis of specifications for the mentioned deficiencies. Basically they reduce a specification to a set of atomic transitions which define an automaton (ANF) or a number of parallel automata (PNF). For the integration of the core system with the adaptation sub-system and finally for the interface abstraction we need the big step semantics. Therefore, we propose the following (iterated) steps to derive the appropriate semantics for each analysis:

1. Formalize use-cases as services
2. Compose services with respect to their relationship and contexts
3. Syntactically transform the specification into the desired normal form
4. Derive the small step semantics
5. Apply the conflict and completeness analysis
6. Supplement the specification
7. Derive the big step semantics
8. Apply any additional analysis (e.g., for safety properties)
9. Derive the system specification by applying the interface abstraction

The benefit of this approach is obvious: The core system is analyzed by means of standard proof systems. Only after removing the two main deficiencies the more complex big step semantics is necessary to analyze the interaction between the core system and the adaptation subsystem and to derive the final interface specification.

6.4. Related work

Throughout this chapter we already mentioned a number of approaches that are close to the ideas of this thesis or had an influence on them. Some of them are close with respect to the semantics, others with respect to mode concepts, or with respect to the usage of the notion of services. We now discuss those approaches that provide concepts for the analysis of specifications both for services in general and for context-adaptive systems in special.

In Section 4.8 we already presented the approach of Harhurin [Harhurin, 2010] for specifying service specifications and described differences to MARLIN. The approach of Harhurin describes a profound concept for the analysis of specifications as well.

The definition of conflicts is equal to the definition in this thesis with the respective changes according to differences in the semantic domain. The analysis for conflicts bases on the extension of services with a failure state that the service takes if it cannot process inputs. An inconsistency exists if a compound service is able to reach a failure state but the sub-services do not reach the failure state for the same inputs (respectively their projection to the sub-services' interfaces).

This approach resembles the idea of the analysis in Section 6.3.1 for specifications in PNF. However, we do not explicitly introduce a failure state. It is sufficient to identify states that correspond (i.e., match on the shared resources) and inputs exist that are processable only by one of the services.

In his publications, Schätz also describes methods for analyzing service specifications. In [Schaetz, 2002] and [Schätz and Salzmann, 2003] he checks consistency and completeness by means of the relational μ-calculus [Bradfield and Stirling, 2001]. Later he changes the formalism to include alternative composition in [Schätz, 2009] and gets closer to ideas presented in [Henzinger, 2000] with parallel and alternative composition. At the same time he changes the analysis of specifications to monadic second order logic [Janin and Walukiewicz, 1996]. Basically the change of paradigms allows using a different proof mechanism, namely Mona [Klarlund and Møller, 2001].

Schäfer presents the integration of formal verification with the model based specification of context-adaptive systems [Schaefer, 2008]. Her approach bases on MARS (Methodologies and Architectures for Runtime Adaptive Systems) which is a modeling technique similar to Trapp's Cameleon [Trapp, 2005]. A mathematical system model called synchronous adaptive systems (SAS) defines the semantic domain in the approach. Basically SAS is a state transition system which distinguishes between state variables used for adaptation decisions and state variables that belong to the core functionality.

A logic called $\mathcal{L}_{SAS}$ (which is a variant of CTL*) allows specifying properties of context-adaptive systems. Proofs are carried out by translating models into, e.g., the input language of Isabelle/HOL [Nipkow et al., 2002] (shallow embedding). This is similar to the approach of Spichkova [Spichkova, 2007] who embedded FOCUS into Isabelle/HOL. Schäfer's approach is specific to the used model for context-adaptive systems. In addition, Schäfer discusses techniques for the reduction of proof complexity like model transformations, slicing and compositional verification.

Compared to our approach this is a more detailed consideration of proofs but it is specific for the MARS specification technique. MARS is a component specification technique. Schäfer's goal is to proof properties of a component specification with respect to an already existing specification rather than analyzing the specifications themselves. MARLIN is capable of providing those properties and ensuring that they are consistent.

Basler introduces ITL+ to apply symbolic execution of parallel programs [Balser, 2005]. ITL+ bases on Interval Temporal Logic (ITL)[Moszkowski, 1985], and allows the nesting of temporal formulas and parallel programs. Nafz uses ITL+ to estab-

lish a formal framework for the verification of organic computing systems [Nafz et al., 2010]. Systems are specified in an assumption/guarantee [Abadi and Lamport, 1995] like manner: as long as the environment meets the assumption the system keeps its guarantees. The semantics uses an alternating state trace: system and environment edit the state alternately. Therefore, the environment must be modeled explicitly.

ITL+ is used to specify the system and a property that is called restore invariant approach. All states that are productive, i.e, do not require a reconfiguration fulfill a certain invariant. Violating the invariant indicates the need for a reconfiguration. The ability of ITL+ to consider parallel specifications allows establishing the organic design pattern which separates the system into an observer/controller (o/c) and a functional layer that are composed in parallel.

From the view of the o/c layer the functional layer is in the environment and vice versa. This opens up for a compositional proof schema that allows a separate consideration of the functional layer and the o/c layer. The idea is similar to the ideas of using the mode normal form to separate the core system from the adaptation subsystem. However, the approach is tailored for organic computing systems. In general, for context-adaptive systems no (feasible) invariant can be established whose violation indicates the need for a reconfiguration. Furthermore, the violation of an invariant does not yet indicate a proper next configuration in general.

Chapter 7

Conclusion and outlook

The topic of this thesis is the specification and the analysis of context-adaptive systems. In this final chapter we close the circle to the beginning and discuss the results with respect to the initial challenge in Section 7.1. Furthermore, we give an outlook to further topics in Section 7.2.

Contents

7.1. Summary

Context-adaptive systems shall relieve users from interactions by means of automation. The automation is desirable if e.g. users are just overstrained, or if it is impossible for users to control the system on their own. The automation is supported by the consideration of additional information whose interpretation is significant for the context of the system usage.

The correct and appropriate behavior of the automation is one of the key success factors of context-adaptive systems. However, adding automation is equivalent to adding additional functions to a yet complex system. This rises the level of complexity and exacerbates the challenges of developing systems.

7.1.1. Achievements

In our analysis of context-adaptive systems in Chapter 3, we define context-adaptive systems by projections on a user interface and a context interface respectively. We claim that a distinction of context-adaptive systems from other systems is artificial and

depends on the user model. However, there is a methodological reason for this distinction.

Systems with a considerable level of automation for selecting functions require additional measures to counteract complexity that is introduced by the automation. For specifying context-adaptive systems the nature of context-adaptive systems offering different functions in different situations provides a starting-point to apply a separation of concerns. Using the view of context-adaptive systems separates the logic of selecting functions from the specification of functions.

A specification describes all implementations that are acceptable [Heitmeyer et al., 1996]. From a logical point of view, any valid implementation is a model of the specification. The existence of a model depends on the consistency of the specification. Since a formal specification is created from the requirements and needs that different stakeholders express, it is unlikely that it is already flawless. Usually it contains deficiencies like inconsistencies or incompleteness. Therefore, we discuss possible methods of the analysis of specifications.

In this thesis we present the MARLIN specification technique consisting of the MARLIN specification language and related methodological aspects that guide through the specification and analysis of an initial formal specification. In the introduction we claim some properties of a proper specification technique for context-adaptive systems:

Formal foundation The thesis presents the MARLIN language that bases on a number of approaches, most notably the functions of Schätz [Schätz, 2009] and ideas taken from Broy's JANUS-approach [Broy, 2005, 2010]. As a basis, services formalize interactions with the system.

The language provides an abstract syntax and suggests a number of concrete syntaxes. Furthermore, it has a formal semantics in the style of Kahn's big step or natural semantics [Kahn, 1988]. In addition, a small step semantics allows an operationalization. We developed the language carefully to ensure a number of algebraic properties and a compositional semantics.

As a characteristic of projections, shared variables and channels are able to model conflicting behaviors. This is necessary to allow a formalization of possibly conflicting requirements with the goal to analyze them, reveal deficiencies, and proof properties.

Explicit modeling of the adaptation logic MARLIN introduces mode transition systems to allow specifying different services and relating them with situations where they shall be available. A powerful mechanism to change modes allows a mapping of complex situations that are indicated by combinations of events. Mode transition systems additionally can establish system modes to handle feature interactions.

The direct mapping of situations into mode transition systems together with the provided transformation into a normal form allows a clean separation of the context-adaptive aspects and the offered services. This supports a deeper understanding of the context of functions. The appropriateness of adaptations can

be validated and possibly discussed apart from the formalization of other system functionality.

Managing Complexity The language allows a decomposition of complex systems into a number of services. Composition operators combine simple services into more complex ones. This allows focusing on various aspects of the system's behaviors while ignoring other yet unnecessary details.

The composition operators relate the services in different ways. Therefore, the composition of complex services can be carried out in different ways according to the different views of the stake holders and their representation in the requirements.

Especially the mode transition systems support a separation of concerns. They allow modeling the basic functions that the system provides apart from the logic that controls the availability of functions. Feature interactions that contribute to the complexity of systems can be handled in a structured way either by mode transition systems or by alternative composition, depending on the nature of the feature interaction.

Appropriate level of abstraction An initial specification should restrict as less implementation details as possible. MARLIN bases on the notion of services that abstracts from structural details of the system. Furthermore, the specification acts on logical events rather than on technical signals. Therefore, it is possible to abstract from the actual acquisition of information and to focus on the occurrence of discrete events. Details about sensors together with the interpretation of their data and actuators together with details of their control are yet out of scope. This allows the re-use of the specification for different technologies.

Methodological aspects Services decompose systems with respect to the interface and interactions with the system that can be observed at these interface clippings. This results in a hierarchic structure. The service hierarchy in MARLIN differs from other hierarchies (like, e.g., [Broy et al., 2007] and [Gruler and Meisinger, 2009]). A number of distinct relations (according to the operators of the language) are possible between children and parental nodes. We describe the characteristics of a service hierarchy formally and guide the development of a service tree that merely captures the existence of relations into a full service hierarchy that captures the nature of the relations. We present unbundling as an approach to integrate requirements with different, possibly overlapping contexts of effectiveness into mode transition systems.

The nature of services as projections on the system's behavior includes a shared use of resources. This coincides with use-cases as a common way to capture requirements form different points of view. On the other hand, this may lead to inconsistencies. The use of certain normal forms and analysis techniques, allows analyzing service hierarchies for remaining deficiencies like inconsistencies and

partiality. Alternative composition and mode transition systems are means to complement such identified inconsistencies in a service hierarchy.

7.1.2. Discussion

As with any formalism MARLIN is tailored for a certain purpose. This purpose is to allow generating consistent specifications of context-adaptive systems by iteratively analyzing and altering the specification. A focus is on enabling and easing the analysis of specifications. As a consequence, some trade-offs are necessary to ensure the algebraic properties.

Preemptive behaviors One trade-off was the ultimate choice of making all services preemptive. This allows distributing parallel composition over mode transition systems. For some systems preemption may not be reasonable. We can easily think of a formalism that is similar to MARLIN but allows run-to-completion. However, the mode normal form as presented does not exist then. Identifying the services that can be active simultaneously and possibly interfere becomes harder without the normal form. As a drawback in MARLIN, patterns are necessary to model non-preemptive behavior.

Shared variables Another choice that can be challenged, is the use of shared variables. Information hiding (especially concerning the internal state) is a well-established paradigm to rise the modularity and reusability of entities. We argue that services are overlapping projections on a system's behavior. If variables serve as auxiliary means to describe the behavior they have to be included in the (overlapping) projections. Harhurin describes a service based specification technique that uses a strict exclusiveness of variables [Harhurin, 2010]. This leads to a formalism that separates aspects of services that model commonalities of views. Influences of services can only be modeled in priority automata – especially because feedback is disallowed as well. In cases where intermediate results of one service shall be available to another one, the specification easily becomes complex.

MARLIN allows describing phases of execution with services disappearing in some phases leaving the control of the system's state to other services. Therefore, it is necessary to allow the manipulation of state variables by all service. Variables in early specifications have an auxiliary character and the interface abstraction finally hides them. They are no proof obligation for the subsequent development artifacts. Information hiding can be applied as usual in the architectural phase.

Expressive power A balance between expressive power and a possibly error prone degree of freedom is necessary. We take the position that a formalism that is capable of formalizing requirements needs much expressive power. This allows an easy formalization of existing requirements together with a subsequent analysis rather than cramming requirements into a restricted formalism. Formalizing requirements with overlapping aspects in a component specification technique is error

prone, because the requirements need to be already altered to be free from conflicts bearing the risk of hiding possible conflicts in their original phrasing.

MARLIN is expressive and gives the system designer various options to formalize requirements. Of course this has the danger to introduce errors that are not inherent to the original requirements phrasing. However, the analysis allows to reveal errors regardless if they were already present in the original phrasing or introduced during their transformation into the formal system.

7.2. Outlook

This thesis discusses only a limited set of aspects. There are a number of closely related topics that are interesting.

Integration with other requirements descriptions: As a starting point we clearly focused on the contextual requirements chunks of Sitou [Sitou, 2009]. The transition of the contextual requirements chunks into MARLIN bases on the already clear separation of requirements and contexts. However, there are other approaches for capturing requirements for context-adaptive systems. To integrate MARLIN with other approaches we need to investigate requirements without such clear separation of contexts.

Additional methodological guidance for unbundling: We introduced unbundling to integrate requirements. Applied to subsets of requirements unbundling creates complex functions out of simpler ones, taking into account the contexts of the integrated requirements. We provided only little methodological support that guides the use of such generated mode transition systems in a hierarchy of mode transition systems, yet.

Patterns: Common constructs for context-adaptive systems need correspondences in the formal language. Some of them are integrated into the MARLIN-language like sharing variables between subsequent behaviors. Reset- and persistent patterns cover other scenarios. There are more constructs to cover by patterns. An example is the integration of mode transition systems that require an update of the mts-variables while the mode transition system is suspended, etc.

Transfer to architecture: As soon as a specification becomes mature enough, one starts defining an architecture. The architecture maps functions to components in a m:n manner using additional information like architecture related requirements and non-functional requirements. Cause-effect-chains relate existing architectures and functions. However cause-effect-chains are an a-posteriori relation. As an open issue, we need a constructive approach that transforms a behavior specification into an architecture specification. Today transformations from services to (good) architectures, preferably using syntactic transformations and supplementations, are rare. There are approaches like [Harhurin, 2010] and [Leuxner et al.,

2010] that head into this direction, but they do not fit to context-adaptive systems nor do they return an optimized architecture with respect to given criteria.

Besides these methodological aspects, an empirical survey on a broad basis for the usability of the approach is required. A small number of successful case studies is not significant for the applicability of an approach. This evaluation includes the creation of a tool prototype. During the development of the thesis we used a number of existing tools like COLA and AutoFOCUS to model certain aspects of case studies with respect to our ideas. However, none of the tools integrates all of the ideas of MARLIN yet.

Appendix A

Properties of MARLIN

In this appendix we summarize the most important algebraic properties of MARLIN and give proofs or proof sketches to underline our arguments.

Contents

A.1. Causality

In a causal specification any outputs depend only on inputs until the present time (weak causality) or up to the time interval before (strong causality). This property is essential for a realizable specification (see, e.g., [Broy and Stølen, 2001]).

To discuss this property for MARLIN we first adapt the property defined in Definition 10 to the semantic domain of MARLIN.

DEFINITION 49 (STRICT CAUSALITY IN THE SEMANTIC DOMAIN OF MARLIN):
We call an I/O-behavior F weakly causal or simply causal, iff

$$\forall\, i_1, i_2 \in \vec{I}, \sigma \in \Sigma, t \in \mathbb{N}:$$
$$(i_1)_{\downarrow t} = (i_2)_{\downarrow t} \Rightarrow \{(o)_{\downarrow t} \mid (o, \acute{\sigma}) \in F.(\sigma, i_1)\} = \{(o)_{\downarrow t} \mid (o, \acute{\sigma}) \in F.(\sigma, i_2)\}$$

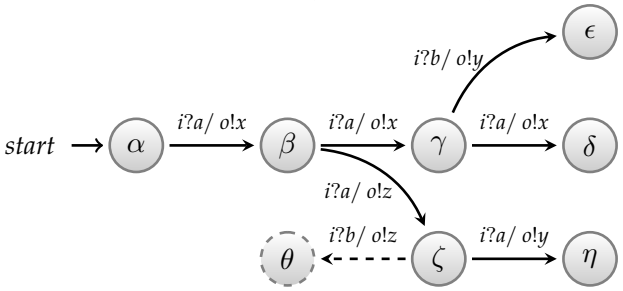
i.e. outputs at time t depend only on inputs until time t. We call the I/O-behavior strictly causal, iff

$$\forall\, i_1, i_2 \in \vec{I}, \sigma \in \Sigma, t \in \mathbb{N}:$$
$$(i_1)_{\downarrow t} = (i_2)_{\downarrow t} \Rightarrow \{(o)_{\downarrow t+1} \mid (o, \acute{\sigma}) \in F.(\sigma, i_1)\} = \{(o)_{\downarrow t+1} \mid (o, \acute{\sigma}) \in F.(\sigma, i_2)\}$$

i.e. outputs at time $t + 1$ depend only on inputs until time t. This fits to calculations requiring some time to happen. □

In general, incomplete MARLIN specifications are not (strictly) causal. Usually the cause is not that these specifications react to future inputs. It is merely a side effect of the combination of non-determinism and underspecification. For illustration refer to Example A.1

EXAMPLE A.1
Let $F\ (i \rhd o, s)$ be a simple specification with $type(i) = \{a, b\}$, $type(o) = \{x, y, z\}$ and $type(s) = \{\alpha, \beta, \gamma, \delta, \epsilon, \zeta, \eta, \theta\}$ and the (graphical) specification



For the moment we ignore the dashed transition. It is easy to see that $F.\langle\!\langle \langle a\rangle\langle a\rangle\langle a\rangle\langle \bullet\rangle \rangle\!\rangle = \{\langle\!\langle \langle\bullet\rangle\langle x\rangle\langle x\rangle\langle x\rangle \rangle\!\rangle, \langle\!\langle \langle\bullet\rangle\langle x\rangle\langle z\rangle\langle y\rangle \rangle\!\rangle\}$ and $F.\langle\!\langle \langle a\rangle\langle a\rangle\langle b\rangle\langle \bullet\rangle \rangle\!\rangle = \{\langle\!\langle \langle\bullet\rangle\langle x\rangle\langle x\rangle\langle y\rangle \rangle\!\rangle\}$. The prefixes for the inputs up to time $t = 2$ are identical. However, in the set of outputs of F to the first input stream an output stream exists that is invalid for the second input – even its prefix until time $t + 1$.

If we include the dashed transition we see that the situation changes: $F.\langle\!\langle \langle a\rangle\langle a\rangle\langle b\rangle \rangle\!\rangle = \{\langle\!\langle \langle\bullet\rangle\langle x\rangle\langle x\rangle\langle y\rangle \rangle\!\rangle, \langle\!\langle \langle\bullet\rangle\langle x\rangle\langle z\rangle\langle z\rangle \rangle\!\rangle\}$. Now the identical prefixes of the inputs produce identical prefixes with respect to possible sets of outputs.

Note that this example is not input complete i.e., underspecified. ♣

As we can see from the example a MARLIN specification easily violates the original property of strong causality. However, the use of MARLIN is to compose partial functions with the opportunity to analyze them. As long as the specification is not finalized, we want to accept violations of strict causality. Therefore, we define a different property that we call restricted causality, that is a necessary condition for strong causality and captures the aspects of strong causality that system reactions need at least one time interval and never depend on future inputs. Together with input enabledness, restricted causality will ensure strong causality in a finalized specification.

DEFINITION 50 (RESTRICTED CAUSALITY):
We call a behavior restricted causal, iff

$$\forall i_1, i_2 \in \vec{I}, t \in \mathbb{N}, \sigma \in \Sigma :$$
$$(i_1)_{\downarrow t} = (i_2)_{\downarrow t} \Rightarrow$$
$$\{o \mid (o, \acute{\sigma}) \in F((i_1)_{\downarrow t+1}, \sigma)\} = \{o \mid (o, \acute{\sigma}) \in F((i_2)_{\downarrow t+1}, \sigma)\}$$

i.e. the outputs at time $t+1$ only depend on inputs up to time t: a difference between i_1 and i_2 at time $t+1$ has no influence on the valid outputs. In contrast to strong causality, restricted causality cares not about any possible blocking in future. □

Coming back to Example A.1, the specification extended with the dashed transition is non-deterministic as well and is not input complete but it is strictly causal. Even more, the non-determinism no longer influences the systems capability of reacting to inputs. Hence, a non-deterministic choice can never be wrong in the sense that the system will block for a certain choice later whereas a different choice allows further reactions.

The situation is closely related to observations typically made for semantic domains of process algebras. To establish a congruence some process algebras like CSP [Hoare, 1985] and ACP [Bergstra and Klop, 1985] use ready sets [Bergstra et al., 1988]. Those sets capture a specification's capabilities to react to inputs for every state. However, in our context, we are not dealing with congruencies. Therefore, we will adapt this notion. We define a property that we call "readiness complete" to address the former issue.

DEFINITION 51 (READINESS COMPLETE):
First we define an auxiliary function

$$ready : \mathbb{F} \times \Sigma \to \mathbb{H}_1$$
$$ready_F(\sigma) = \{i \in \mathbb{H}_1(I_F) \mid \exists o \in \mathbb{H}_1(O_F), x \in \vec{I}, y \in \vec{O}, \acute{\sigma} \in \Sigma_F :$$
$$(\sigma, x, y, \acute{\sigma}) \in F \ \wedge x.1 = i \wedge y.2 = o\}$$

that returns the set of accepted next inputs for a state σ. We say that a service F is readiness complete iff

$$\forall (\sigma, i_1, o_1, \acute{\sigma}_1), (\sigma, i_2, o_2, \acute{\sigma}_2) \in [\![F]\!] : i_1 = i_2 \Rightarrow ready_F(\acute{\sigma}_1) = ready(\acute{\sigma}_2)$$

i.e., if starting in the same state and reading the same inputs, the specification allows reactions to the same set of next inputs, regardless of the executed "path" in the transition system. □

Readiness completeness extends restricted causality. For a restricted causal service F that is readiness complete the following more general property holds (without proof):

$$\forall(\sigma, i_1), (\sigma, i_2) \in \operatorname{dom} F, t \in \mathbb{N} :$$
$$(i_1)_{\downarrow t} = (i_2)_{\downarrow t} \Rightarrow \{(o)_{\downarrow t+1} \mid (o, \acute{\sigma}) \in F.(\sigma, i_1)\} = \{(o)_{\downarrow t+1} \mid (o, \acute{\sigma}) \in F.(\sigma, i_2)\}$$

Note that this is still weaker than restricted causality because the property only holds for all inputs in the domain of F and not for all possible inputs.

In general the semantic domain of MARLIN deals with streams and faces the issues of causality as described in [Broy and Stølen, 2001]. However, we build behaviors from states and state transitions. Each of the state transitions and their outputs only depend on the current state. Therefore, outputs in specifications in MARLIN never will depend on future inputs or outputs i.e. all MARLIN specifications are restricted causal. The only issue is that choosing a certain output (i.e., making a non-deterministic choice) may result in a block in the future whereas making another choice may allow continuing the behavior.

THEOREM 11 (RESTRICTED CAUSALITY OF MARLIN SPECIFICATIONS)
Every MARLIN *specification is restricted causal.*

PROOF 11.1 (OF THEOREM 11):
It is not very difficult to see that the property holds. The property basically describes big steps as they are defined by a Moore automaton [Moore, 1956] as well. Adding or removing transitions in Moore automata never violates the property. Therefore we only outline the proof for MARLIN. *The proof bases on an induction on the structure of a* MARLIN *specification.*

BASE CASE: *Atomic services are built by single transitions. The transitions base on Moore automata Moore [1956] i.e. outputs never depend on current inputs but only on the current state. The current state is determined by earlier inputs only. The definition of the atomic services shows this very well:*

$$pre_i(\sigma) \wedge inpattern_i(c_{\lfloor I}.1) \wedge outpattern_i(\sigma, c_{\lfloor I}.1, c_{\lfloor O}.2) \wedge post_i(\sigma, c_{\lfloor I}.1, \acute{\sigma})$$

To proof that the closure maintains the property we use a second induction. To avoid ambiguities we change the indexing of the inputs of streams that are equal up to a time t from i_1 and i_2 to i and i′ and use the indexing i_1 and i_2 for consecutive parts of a stream only (similar to the output streams).

The closure operation concatenates behaviors with common initial and final states respectively.

$$\begin{aligned} Y = \ \mu Y.X \cup \{(\sigma, & i_1 \frown i_2, o_1 \frown o_2, \acute{\sigma}) \mid \exists \sigma_l \in \Sigma, t \in \mathbb{N} \\ & (\sigma, \ i_1, \ o_1, \ \sigma_l) \in Y \wedge \\ & (\sigma_l, \ i_2, \ o_2, \ \acute{\sigma}) \ \in Y \ \} \end{aligned}$$

This is an iterative application of a function: $\leftrightharpoons$: $[\![F]\!] \times [\![F]\!] \to (\Sigma_F \times \vec{I_F} \times \vec{O_F} \times \Sigma_F)$, *such that*

$$[(\sigma_1, i_1, o_1, \sigma_2) \leftrightharpoons (\sigma_3, i_2, o_2, \sigma_4) \mapsto (\sigma_1, i_1 \frown i_2, o_1 \frown o_2, \sigma_4)] \Leftrightarrow \sigma_2 = \sigma_3$$

BASE CASE OF INNER INDUCTION: *Again the base case is a set of behaviors that only contains single transitions that trivially exhibit the property.*

INDUCTION HYPOTHESIS OF INNER INDUCTION *We assume that a set of behaviors that is subject to the application of one closure iteration is restricted causal.*

INDUCTION STEP OF INNER INDUCTION: *In the induction step we investigate one iteration of the application of the function* $\leftrightharpoons$. *Let* $[\![\grave{F}]\!]$ *be the set before the iteration and* $[\![\acute{F}]\!]$ *the set after the iteration. For the induction we assume that the property holds for all elements in* $[\![\grave{F}]\!]$

We investigate arbitrary i and i' for all t and distinguish two cases:

1. $(i)_{\downarrow t} = (i')_{\downarrow t}$ *such that* $\exists(\sigma, (i)_{\downarrow t}, o, \acute{\sigma}) \in [\![\grave{F}]\!]$ *the property obviously holds due to the induction hypothesis.*
2. $(i)_{\downarrow t} = (i')_{\downarrow t}$ *such that* $\exists(\sigma, (i)_{\downarrow t}, o, \acute{\sigma}) \in ([\![\acute{F}]\!] \backslash [\![\grave{F}]\!])$ *i.e. the behavior is newly generated by the iteration. In this case we know that*

   $$\{o \mid (o, \acute{\sigma}) \in F((i)_{\downarrow t+1}, \sigma)\}$$

   is created by

   $$[\![\acute{F}]\!] = [\![\grave{F}]\!] \cup \{(\sigma, i_1 \frown i_2, o_1 \frown o_2, \acute{\sigma}) \mid \exists\, \sigma_l \in \Sigma, t \in \mathbb{N} \\ (\sigma,\ i_1,\ o_1,\ \sigma_l) \in [\![\grave{F}]\!] \wedge \\ (\sigma_l,\ i_2,\ o_2,\ \acute{\sigma}) \in [\![\grave{F}]\!]\ \}$$

   and

   $$\{o \mid (o, \acute{\sigma}) \in F((i')_{\downarrow t+1}, \sigma)\}$$

   is created by

   $$[\![\acute{F}]\!] = [\![\grave{F}]\!] \cup \{(\sigma, i'_1 \frown i'_2, o'_1 \frown o'_2, \acute{\sigma}) \mid \exists\, \sigma_l \in \Sigma, t \in \mathbb{N} \\ (\sigma,\ i'_1,\ o'_1,\ \sigma_l) \in [\![\grave{F}]\!] \wedge \\ (\sigma_l,\ i'_2,\ o'_2,\ \acute{\sigma}) \in [\![\grave{F}]\!]\ \}$$

   According to the induction hypothesis:

   $$\{(\sigma, i_1, o_1, \sigma_l) \mid (\sigma, i_1, o_1, \sigma_l) \in [\![\grave{F}]\!]\} = \{(\sigma, i'_1, o'_1, \sigma_l) \mid (\sigma, i'_1, o'_1, \sigma_l) \in [\![\grave{F}]\!]\}$$

   for $i_1 = i'_1$ *and*

   $$\{(\sigma_l, (i_2)_{\downarrow t+1}, (o_2)_{\downarrow t+1}, \acute{\sigma}) \mid (\sigma_l, (i_2)_{\downarrow t+1}, (o_2)_{\downarrow t+1}, \acute{\sigma}) \in [\![\grave{F}]\!]\} = \\ \{(\sigma_l, (i'_2)_{\downarrow t+1}, (o'_2)_{\downarrow t+1}, \acute{\sigma}) \mid (\sigma_l, (i'_2)_{\downarrow t+1}, (o'_2)_{\downarrow t+1}, \sigma_l) \in [\![\grave{F}]\!]\}$$

for all t with $i_2\lfloor_t = i'_2\lfloor_t$. *From that we conclude that*

$$\begin{aligned}\forall\, t : i\lfloor_t = i'\lfloor_t \Rightarrow \\ \{o_1 \frown o_2 \mid (o_1 \frown o_2, \acute{o}) \in F((i_1 \frown i_2)_{\downarrow t+1}, \sigma)\} = \\ \{o'_1 \frown o'_2 \mid (o'_1 \frown o'_2, \acute{o}) \in F((i'_1 \frown i'_2)_{\downarrow t+1}, \sigma)\}\end{aligned}$$

Together the two cases show that the closure operation maintains restricted causality.

INDUCTION HYPOTHESIS: *The operands of any composition operator are restricted causal.*

INDUCTION STEP: *We investigate the three composition operators.*

Alternative composition: *Similar to the base case. The closure operation maintains the property as demonstrated already for the atomic services.*

Parallel composition: *Parallel composition adds no new behaviors (it does not extend the system with additional capabilities to react to inputs if in a certain state). The hull operation only removes behaviors if they are not consistent with interruption (cf. Section A.2). Removing behaviors affects both sets in the property definition (the set for i and i′) in the same way.*

Conditional composition: *Similar to alternative composition. Conditional composition adds additional information to the states such that only some states qualify as common states i.e. the mode and history pointer must match. Additionally a pasting is only possible if an appropriate transition condition holds. However, the pasting then finally works analogous to the closure operation in the alternative composition. This becomes obvious if we have a look at the structure of the defining equation of the conditional composition that resembles the structure of the closure operation.*

$$\begin{aligned}\{(\sigma, i, o, \acute{\sigma}) \mid \exists \ldots \sigma_l \in \Sigma \ldots : \\ \ldots \\ \wedge\, (\sigma, (i)_{\downarrow t+1}, (o)_{\downarrow t+1}, \sigma_l)\lfloor_{\Phi_m} \in (\llbracket \Phi_m \rrbracket \cup \llbracket ANY_1 \rrbracket) \qquad (5) \\ \ldots \\ \wedge\, (\sigma_l, (i)^{\uparrow t}, (o)^{\uparrow t}, \acute{\sigma}) \in \llbracket F_{m'} \rrbracket\} \qquad (7)\end{aligned}$$ q.e.d.

According to the proof obligations in Section 6.1, we are finally interested in complete specifications only. Therefore, readiness completeness is no issue during the specification activities. In Section 6.3.2 we present a method to check a specification for input completeness. Input complete specifications are by definition readiness complete ($\forall\, \sigma \in \Sigma : ready_F(\sigma) = \mathbb{H}_1(I_F)$) and are a necessary condition for realizable specification (see [Broy and Stølen, 2001] as well).

A.2. Consistency with interruption

Before we discuss algebraic properties of MARLIN we introduce a set of general properties. Some of the proofs for the algebraic properties base on these properties. Informally spoken, these properties ensure that a behavior can be interrupted and resumed at any time and interrupting a behavior and immediately resuming it does not introduce new behaviors.

While this property seems obvious it is initially uncertain that the compositions preserves this property. We illustrate this with an example.

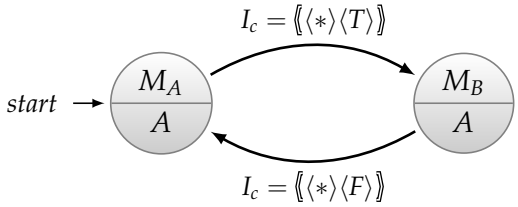
EXAMPLE A.2 (INCONSISTENCY WITH RESUMPTION)
The subsequent example is artificial because we will see later that MARLIN *maintains the property. We only consider one input and one output channels each.*

[A]

in	$i : \{\alpha, \beta\}$
out	$o : \{\alpha, \beta, \gamma\}$
var	$s : \{1, 2, 3, 4\}$

$(1, \{\!\{\langle\alpha\rangle\langle\bullet\rangle\}\!\}, \{\!\{\langle\bullet\rangle\langle\alpha\rangle\}\!\}, 2)$,
$(2, \{\!\{\langle\beta\rangle\langle\bullet\rangle\}\!\}, \{\!\{\langle\bullet\rangle\langle\beta\rangle\}\!\}, 3)$,
$(1, \{\!\{\langle\alpha\rangle\langle\beta\rangle\langle\bullet\rangle\}\!\}, \{\!\{\langle\bullet\rangle\langle\alpha\rangle\langle\beta\rangle\}\!\}, 3)$,
$(2, \{\!\{\langle\beta\rangle\langle\bullet\rangle\}\!\}, \{\!\{\langle\bullet\rangle\langle\gamma\rangle\}\!\}, 4)$

Furthermore, assume a mode transition system F



We compare the service's possible interactions a) with a mode change after one time interval and b) with no mode change.

a) after receiving $\langle\alpha\rangle$ and sending $\langle\alpha\rangle$ the service reaches state ②. Meanwhile the mode transition system interrupts the service. In the next mode the service continues at this state. According to A's possible behaviors this allows sending $\langle\gamma\rangle$ after receiving $\langle\beta\rangle$.

b) The whole I/O-stream needs to be valid for service A. Therefore, receiving $\langle\alpha\rangle$ followed by a $\langle\beta\rangle$ allows only the output of $\langle\alpha\rangle$ followed by $\langle\beta\rangle$

Figure A.1 illustrates the situation for a) and b).

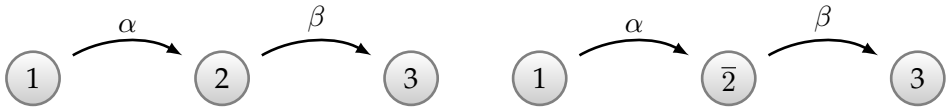
t	1	2	3
I_C	T	$-$	
I	α	β	
O		α	γ

(a) $\in F$

t	1	2	3
I_C	$-$	$-$	
I	α	β	
O		α	γ

(b) $\notin F$

Figure A.1.: Two example runs, a) is in F and b) is not



(a) $\{(1, \alpha, 2), (2, \beta, 3), (1, \alpha \circ_t \beta, 3)\}$ **(b)** $\{(1, \alpha, \overline{2}), (\overline{2}, \beta, 3), (1, \alpha \circ_t \beta, 3)\}$

Figure A.2.: Two simple services with conflicting states 2 and $\overline{2}$, with simplified illustration of the streams

The interruption by the mode transition system enables a behavior that is no allowed behavior without the interruption. A possible cause for the existence of such additional behavior is an incomplete definition of parallel composition. Regard the two services in Figure A.2.

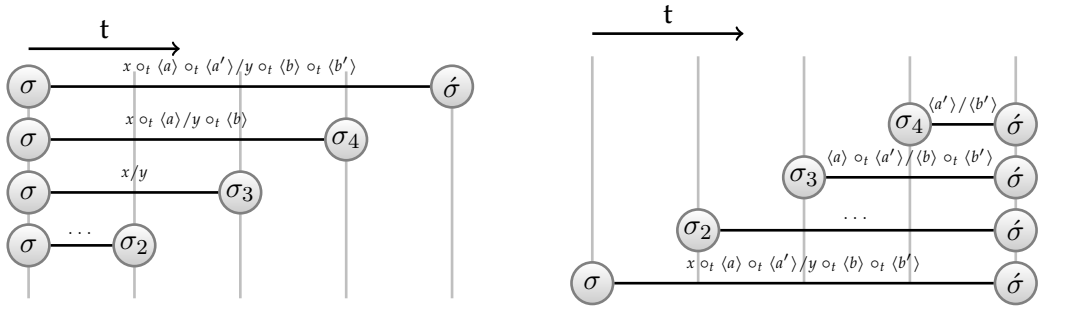
The intersection of the two sets of behaviors contains only the tuple $(1, \alpha \circ_t \beta, 3)$*. Similar to the example above: this can be a valid behavior if the compound behavior is never interrupted. However, there is no intermediate state that satisfies both,* ② *and* $\overline{②}$*. As a result, a mode transition system using the service in two subsequent modes must not change the mode after the first time interval.* ♣

To address the issue, we define a set of properties that summarize to the property of consistency with interruption.

DEFINITION 52 (INFIX MONOTONICITY):
We call a service F infix monotonic iff:

$$\forall(\sigma, i, o, \acute{\sigma}) \in [\![S]\!] : \#(i) > 2 \Rightarrow$$
$$\exists\, i_1, i_2 \in \vec{I_S};\ o_1, o_2 \in \vec{O_S}\, \exists\, \sigma_l \in \Sigma_S :$$
$$i = i_1 \frown i_2 \wedge o = o_1 \frown o_2 \wedge (\sigma, i_1, o_1, \sigma_l) \in [\![S]\!] \wedge (\sigma_l, i_2, o_2, \acute{\sigma}) \in [\![S]\!]$$

We write IFX(S) □



(a) Illustration of prefix monotonicity for a behavior $(\sigma, x \circ_t \langle a \rangle \circ_t \langle a' \rangle, y \circ_t \langle b \rangle \circ_t \langle b' \rangle, \sigma^5)$

(b) Illustration of suffix monotonicity for a behavior $(\sigma, \langle a \rangle \circ_t \langle a' \rangle \circ_t x / \langle b \rangle \circ_t \langle b' \rangle \circ_t y, \sigma^5)$

Figure A.3.: Demonstration of prefix- and suffix monotonicity

Infix monotonicity includes prefix- and suffix monotonicity taking an appropriate intermediated state into account. A service that is infix monotonic defines a set of behaviors such that for each valid behavior a prefix of the I/O-history together with an appropriate intermediate state exists such that the prefix is a valid behavior as well, finishing at the intermediate state. At the same time another valid behavior exists starting in the intermediate state that is according to the rest of the original behavior. By transitivity, this property actually requires that all fragments of a behavior are valid behaviors with according initial and final states as well.

Figure A.3a illustrates prefix monotonicity and Figure A.3b illustrates suffix monotonicity. Suffix monotonicity is similar to prefix monotonicity but concerns the rest of an I/O-history. Note that the intermediate states in both figures coincide by intention.

Infix monotonicity allows interrupting and resuming behaviors at any time. The composition of the services in Figure A.2 violates this property. However, service A in example A.2 satisfies both properties but still does not meet our expectations. The Figure illustrates a situation with a behavior whose initial state corresponds to a final state of another behavior, but their concatenation is no valid behavior. The red colored

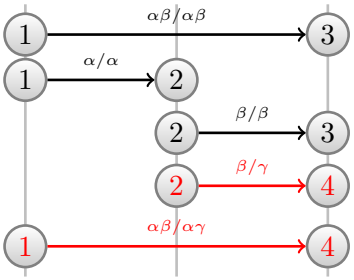


Figure A.4.: Clipping of valid behaviors of a service A. Immediate resumption: together with (σ, x, y, σ_5) and $(\sigma, x_1, y_1, \sigma_2)$ the behavior $(\sigma_2, x_2, y_2, \sigma_5)$ has to be a valid behavior such that $x = x_1 \circ x_2$ and $y = y_1 \circ y_2$

behavior in Figure A.4 illustrates the additional behavior. This shows that infix closure is yet insufficient to restrict behaviors to meet our requirements. All possible assemblies of behaviors need to be valid behaviors of the service as well.

DEFINITION 53 (BEHAVIORAL CLOSURE):
A service S is behavioral closed iff:

$$[\![S]\!] = \mathcal{CL}([\![S]\!])$$

We write $BC(S)$. □

Only both properties together result in the desired property: I) a behavior can be interrupted and immediately resumed in the next mode at any time and II) interrupting and resuming introduces no new behaviors or alters the validity of existing behaviors.

DEFINITION 54 (CONSISTENT WITH INTERRUPTION):
A service S is consistent with interruption iff:

$$IFX(S) \wedge BC(S)$$

We write $CWI(S)$. □

The semantic domain of MARLIN inherently has the danger to violate either of these properties. However, these properties are a prerequisite for the proofs of the algebraic properties and transformations. Therefore, we need to proof that the composition operators of MARLIN maintain these properties.

THEOREM 12
All services that are built up from atomic services that are consistent with interruption by using the composition operators of MARLIN *are consistent with interruption as well.*

PROOF 12.1 (OF THEOREM 12):
We only sketch the formal proof. The proof is by induction on the structure of the service specification: parallel, alternative and conditional composition have to preserve the properties. For each of the cases we use an induction on the length of behaviors.

BASE CASE: ATOMIC SERVICES *Obviously atomic services are consistent with interruption. They are built from single transitions and their closure. The closure operation ensures the property BC. The closure takes every single behavior (especially the single steps) and combines them recursively into longer ones. Obviously for any longer behavior the fragments are in the set of behaviors as well.*

ALTERNATIVE COMPOSITION *As an induction hypothesis we assume that both operands are consistent with interruption. After joining the sets of behaviors the closure operation in the alternative composition combines any two behaviors whose final state of the first one matches to the initial state of the second one. As a result any valid behavior is either due to one of the operands' behaviors or it is an alternation between the operands' behaviors.*

Infix monotonicity *According to the induction hypothesis any behavior of the resulting set of behaviors that belongs to one of the operands' set of behaviors is already infix monotonic. This is the base case for the induction on the length of the streams. Therefore, we only investigate behaviors that are composed from behaviors of both operands by the closure. Without restriction let* $(\sigma, i, o, \acute{\sigma}) \in [\![A \oplus B]\!]$ *be such that*

$$\exists\, \sigma_l \in \Sigma, t \in \mathbb{N}$$
$$(\sigma,\ (i)_{\downarrow t},\ (o)_{\downarrow t},\ \sigma_l) \in [\![A \oplus B]\!] \wedge$$
$$(\sigma_l,\ (i)^{\uparrow t-1},\ (o)^{\uparrow t-1},\ \acute{\sigma}) \in [\![B]\!]\ \}$$

with $(i)^{\uparrow t-1} = x \circ_t \langle x_1 \rangle$ *and* $(o)^{\uparrow t-1} = y \circ_t y_1$ *and* $\#(x \circ_t \langle x_1 \rangle) > 2$*. This decomposition exists because the behavior is an alternation between behaviors of A and B that is created by the closure. The closure needs the appropriate intermediate state in order to paste together behaviors from A and B.*

We split the behavior at the last position where the combined behavior switches from a behavior according to A to a behavior according to B. Figure A.5 illustrates the decomposition.

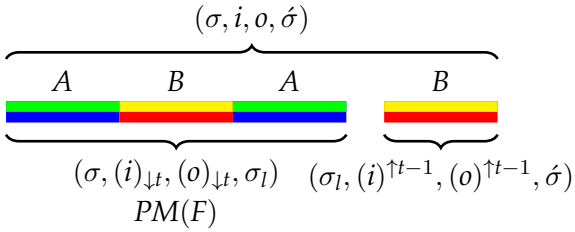


Figure A.5.: Decomposition of a combined behavior in F

By the induction hypothesis we know that $\exists\, \sigma_k : (\sigma_l, x, y, \sigma_k) \in [\![B]\!]$ *and by the use of the closure operation we also know that*

$$(\sigma, i', o', \sigma_k) \in [\![A \oplus B]\!] \text{ such that}$$
$$(\sigma,\ (i')_{\downarrow t},\ (o')_{\downarrow t},\ \sigma_l) \in [\![A \oplus B]\!] \wedge$$
$$(\sigma_l,\ (i')^{\uparrow t-1},\ (o')^{\uparrow t-1},\ \acute{\sigma}) \in [\![B]\!]\ \}$$

with $(i')^{\uparrow t-1} = x$*,* $(o')^{\uparrow t-1} = y$ *and* $(i')_{\downarrow t} = (i)_{\downarrow t}$*,* $(o')_{\downarrow t} = (o)_{\downarrow t}$*. In the special case where* $\#(x \circ_t \langle x_1 \rangle) = 2$ *the truncation of a single time interval already returns a fragment that is valid for A only.*

To this end we showed that stepwise truncation beginning at the last change from one behavior to the other maintains the property. To complete the proof for infix monotonicity we have to show that the same is true for truncations beginning at the first change from one behavior to the other. This proof is almost identical. Therefore we omit this step.

Behavioral closure *By the application of the closure operation*

PARALLEL COMPOSITION *Again we assume that the operands are consistent with interruption. Parallel composition uses no closure operation. As demonstrated above, the intersection of behaviors possibly leads to settings that violate the property of consistency with interruption. Therefore, a kind of inverse function to the closure called $\mathcal{H}$ removes those behaviors that do not satisfy the property.*

Infix monotonicity *The operation $\mathcal{H}$ splits a behavior at an arbitrary position. If the fragments are valid behaviors, the behavior remains in the set – otherwise, it is dropped. Again the operation is applied to the fragments until the fragments are behaviors of length 2. Formally,*

$$
\begin{aligned}
&(\sigma, i, o, \acute{\sigma}) \in [\![F]\!] \Rightarrow \\
&\quad \exists\, \sigma_1, \ldots, \sigma_n \in \Sigma_F, i_1, \ldots, i_{n+1} \in \mathbb{H}_2(I), o_1, \ldots o_{n+1}, \in \mathbb{H}_2(O) : \\
&\qquad (\sigma, i_1, o_1, \sigma_1) \in [\![F]\!] \wedge \\
&\qquad (\bigwedge_{j=1}^{n-1} (\sigma_j, i_{j+1}, o_{j+1}, \sigma_{j+1}) \in [\![F]\!]) \wedge \\
&\qquad (\sigma_n, i_{n+1}, o_{n+1}, \acute{\sigma}) \in [\![F]\!] \wedge \\
&\quad i = i_1 \frown i_2 \frown \cdots \frown i_{n+1} \wedge \\
&\quad o = i_1 \frown o_2 \frown \cdots \frown o_{n+1}
\end{aligned}
$$

Let $F = A \otimes B$. By the definition of the parallel composition we know that

$$
\begin{aligned}
&(\sigma_j, i_{j+1}, o_{j+1}, \sigma_{j+1}) \in [\![A \otimes B]\!] \Rightarrow \\
&\quad (\sigma_j, i_{j+1}, o_{j+1}, \sigma_{j+1})_{\lfloor A} \in [\![A]\!] \wedge \\
&\quad (\sigma_j, i_{j+1}, o_{j+1}, \sigma_{j+1})_{\lfloor B} \in [\![B]\!] \wedge
\end{aligned}
$$

Furthermore, by the assumption about the consistency with interruption (here especially the behavioral closure) of A and B we know that

$$
\begin{aligned}
&\forall\, 1 < r \leq n : \\
&\quad \exists\, x_1, x_2 \in \vec{I_A}, y_1, y_2 \in \vec{O_A} : \\
&\qquad x_1 = i_{1\lfloor A} \frown \cdots \frown i_{r\lfloor A} \wedge y_1 = o_{1\lfloor A} \frown \cdots \frown o_{r\lfloor A} \\
&\qquad \wedge\, (\sigma_{\lfloor A}, x_1, y_1, \sigma_{r\lfloor A}) \in [\![A]\!] \wedge \\
&\qquad x_2 = i_{r+1\lfloor A} \frown \cdots \frown i_{n+1\lfloor A} \wedge y_2 = o_{r+1\lfloor A} \frown \cdots \frown o_{n+1\lfloor A} \\
&\qquad \wedge\, (\sigma_{r\lfloor A}, x_2, y_2, \acute{\sigma}_{\lfloor A}) \in [\![A]\!] \wedge \\
&\quad \exists\, x_3, x_4 \in \vec{I_B}, y_3, y_4 \in \vec{O_B} : \\
&\qquad x_3 = i_{1\lfloor B} \frown \cdots \frown i_{r\lfloor B} \wedge y_3 = o_{1\lfloor B} \frown \cdots \frown o_{r\lfloor B}
\end{aligned}
$$

$$\wedge\, (\sigma_{\lfloor B}, x_3, y_3, \sigma_{r\lfloor B}) \in [\![B]\!] \wedge$$
$$x_4 = i_{r+1\lfloor B} \frown \cdots \frown i_{n+1\lfloor B} \wedge y_4 = o_{r+1\lfloor B} \frown \cdots \frown o_{n+1\lfloor B}$$
$$\wedge\, (\sigma_{r\lfloor B}, x_4, y_4, \acute{\sigma}_{\lfloor B}) \in [\![B]\!]$$

Therefore we conclude

$$(\sigma, x, y, \acute{\sigma}) \in [\![A \otimes B]\!] \Rightarrow$$
$$\exists\, \sigma_l \in \Sigma, x_1, x_2 \in \vec{I}, y_1, y_2 \in \vec{O} : (\sigma, x_1, y_1, \sigma_l) \in [\![A \otimes B]\!] \wedge (\sigma_l, x_2, y_2, \acute{\sigma}) \in [\![A \otimes B]\!]$$

Behavioral closure *Similar as infix closure by the closure of the operands and the property of $\mathcal{H}$ to decompose behaviors into single steps.*

MODE TRANSITION SYSTEMS *Again we assume that the mode-behaviors are already consistent with interruption. The situation is similar to alternative composition. However, rather than sticking together arbitrary matching behaviors mode transition systems paste only behaviors that belong to successive modes and whose I/O streams satisfy a transition condition. Formally:*

Infix monotonicity *For any behavior $(\sigma, i, o, \acute{\sigma})$ with $\#(i) = \#(o) > 2$ we identify a behavior $(\sigma, i_1, o_1, \sigma_l)$ with $i' \sqsubset i, o' \sqsubset o$ and $(\sigma_l, i_2, o_2, \acute{\sigma})$ with $i = i_1 \frown i_2 \wedge o = o_1 \frown o_2$. According to the definition of mode transition systems we distinguish two cases. Let $M = (\mathbb{M}, \delta_M, \Phi_M)$ be the mode transition system.*

a) $\sigma(PC_M) = m \wedge \forall\, t \in [1, \#(i) - 1], (m, (c, l), m') \in \delta : \neg\Theta_F(\sigma, c, (i)_{\downarrow t}, (o)_{\downarrow t})$
According to the definition of mode transition systems the mode transition system controls the whole behavior of $\Phi(m)$. Since this behavior is already infix closed we find a σ_l' such that $(\sigma, i_1, o_1, \sigma_l)_{\lfloor \Phi(m)}, (\sigma_l, i_2, o_2, \acute{\sigma})_{\lfloor \Phi(m)} \in \Phi(m)$ and

$$\sigma_{l\lfloor \Phi(m)} = \sigma_l' \wedge \sigma_l(PC_M) = m \wedge \sigma_l(h_M) = \sigma(h_F) \circ ((i' \bowtie o')_{\lfloor C_F})$$

b) $\sigma(PC_M) = m \wedge \exists\, t \in [1, \#(i) - 1], (m, (c, l), m') \in \delta : \Theta_F(\sigma, c, (i)_{\downarrow t}, (o)_{\downarrow t})$
In that case we find $(\sigma, (i)_{\downarrow t+1}, (o)_{\downarrow t+1}, \sigma_l)_{\lfloor \Phi(m)} \in \Phi(m)$ and $(\sigma_l, (i)^{\uparrow t}, (o)^{\uparrow t}, \acute{\sigma}) \in [\![F_{m'}]\!]$.

According to case a) we know that $(\sigma, (i)_{\downarrow t+1}, (o)_{\downarrow t+1}, \sigma_l)$ is in $[\![M]\!]$ if we choose $\sigma_l(PC_M)$ and $\sigma_l(h_F)$ according to the definition of the mode transition system.

Furthermore we know that $(\sigma_l, (i)^{\uparrow t}, (o)^{\uparrow t}, \acute{\sigma})$ is in $[\![M]\!]$ because $[\![M]\!]$ is defined as the set that unites all behaviors for all modes.

Trivially, infix closure holds for the first part $(\sigma, (i)_{\downarrow t+1}, (o)_{\downarrow t+1}, \sigma_l)$. This is due to case a).

The property holds for the second part $(\sigma_l, (i)^{\uparrow t}, (o)^{\uparrow t}, \acute{\sigma})$ as well. To show that we just need to iteratively apply the argument again. We repeat the truncations until the reminder satisfies case a).

Behavioral closure *This is less obvious compared to the alternative composition because we did not apply an explicit closure operation. Again we distinguish the obvious two cases :*

a) $\sigma(PC_M) = m \wedge \forall\, t \in [1, \#(i) - 1], (m, (c, l), m') \in \delta : \neg\Theta_F(\sigma, c, (i)_{\downarrow t}, (o)_{\downarrow t})$
Due to the assumption that the mode behaviors are behavioral closed this case is trivial.

b) $\sigma(PC_M) = m \wedge \exists\, t \in [1, \#(i) - 1], (m, (c, l), m') \in \delta : \Theta_F(\sigma, c, (i)_{\downarrow t}, (o)_{\downarrow t})$
Looking at the definition of the mode transition systems and the definition of the behavioral closure we see that the structure is quite similar. The mode transition systems definition pastes together all behaviors at matching states:

$$\{(\sigma, i, o, \acute{\sigma}) \mid \exists \ldots \sigma_l \in \Sigma \ldots :$$

$$\ldots$$

$$\wedge\, (\sigma, (i)_{\downarrow t+1}, (o)_{\downarrow t+1}, \sigma_l)_{\lfloor \Phi_m} \in (\llbracket \Phi_m \rrbracket \cup \llbracket ANY_1 \rrbracket) \qquad (5)$$

$$\ldots$$

$$\wedge\, (\sigma_l, (i)^{\uparrow t}, (o)^{\uparrow t}, \acute{\sigma}) \in \llbracket F_{m'} \rrbracket\} \qquad (7)$$

The rest of the mode transition system definition applies additional conditions on the intermediate state and the selection of the successor state. Hence, pasting two behaviors is only possible if they match at $\sigma_l(PC_M)$ as well. This is handled by the mode transition system. However we know that there exist no other behaviors that are candidates for such a pasting. *q.e.d.*

It is important to note that the property of CWI just postulates that an interrupted behavior can be continued immediately in the same way as if it was not interrupted provided that the state is maintained. This only happens if the service under consideration is an element of two consecutive mode behaviors.

A.3. Algebraic Properties

MARLIN is an algebraic structure. Its carrier set is the set of 4-tuples over the local variables, input-, and output streams. The operations of MARLIN are the functions of the algebra. Subsequently we present a number of algebraic properties of MARLIN-expressions. These algebraic properties are useful for

- re-structuring and re-factoring specifications, e.g., for term rewriting, or for deploying them on different components.
- proofing the existence of the analytical normal form as presented in Section 6.2.2.

As already argued in Section 4.3 the decomposition of a system is kind of arbitrary and is heavily influenced by the designers' point of view. Usually domain properties guide a decomposition in addition to technical objectives. This is of great value for understanding the problem domain and the solution space but an efficient analysis requires different structures.

A.3.1. Basic properties

The operators of MARLIN satisfy a number of standard algebraic properties that allow reorganizing and restructuring a service expression. Additionally, they allow a certain degree of freedom in the order of composition.

THEOREM 13 (COMMUTATIVITY LAWS)

$$F_1 \otimes F_2 = F_2 \otimes F_1 \qquad \textit{Commutativity of } \otimes \qquad \text{(A.1)}$$

$$F_1 \oplus F_2 = F_2 \oplus F_1 \qquad \textit{Commutativity of } \oplus \qquad \text{(A.2)}$$

PROOF 13.1 (OF THEOREM 13 EQUATION (A.1)):
Trivially by the commutativity of $\wedge$ in the definition of $\otimes$

$$\begin{aligned}
[\![A \otimes B]\!] \stackrel{def}{=} \mathcal{H}.\{\ & (\sigma, i, o, \acute{\sigma}) \mid \exists z \in \mathbb{H}(C) : \\
& z\lfloor_{\underline{I}} = i \wedge z\lfloor_{O} = o \quad \wedge \\
& (\sigma\lfloor_{\underline{V_A}}, z\lfloor_{\underline{I_A}}, z\lfloor_{O_A}, \acute{\sigma}\lfloor_{\underline{V_A}}) \in [\![A]\!] \quad \wedge \\
& (\sigma\lfloor_{\underline{V_B}}, z\lfloor_{\underline{I_B}}, z\lfloor_{O_B}, \acute{\sigma}\lfloor_{\underline{V_B}}) \in [\![B]\!]\ \} \\
= \\
& \mathcal{H}.\{\ (\sigma, i, o, \acute{\sigma}) \mid \exists z \in \mathbb{H}(C) : \\
& z\lfloor_{\underline{I}} = i \wedge z\lfloor_{O} = o \quad \wedge \\
& (\sigma\lfloor_{\underline{V_B}}, z\lfloor_{\underline{I_B}}, z\lfloor_{O_B}, \acute{\sigma}\lfloor_{\underline{V_B}}) \in [\![B]\!] \quad \wedge \\
& (\sigma\lfloor_{\underline{V_A}}, z\lfloor_{\underline{I_A}}, z\lfloor_{O_A}, \acute{\sigma}\lfloor_{\underline{V_A}}) \in [\![A]\!]\ \} = [\![B \otimes A]\!]
\end{aligned}$$

q.e.d.

PROOF 13.2 (OF THEOREM 13 EQUATION (A.2)):
Similar to Proof 13.1 *q.e.d.*

THEOREM 14 (ASSOCIATIVITY LAWS)

$$(A \otimes B) \otimes C = A \otimes (B \otimes C) = A \otimes B \otimes C \tag{A.3}$$
$$(A \oplus B) \oplus C = A \oplus (B \oplus C) = A \oplus B \oplus C \tag{A.4}$$

PROOF 14.1 (OF THEOREM 14 EQUATION (A.3)):
By applying the definition of parallel composition we get:

$$I_{(A,B)} = I_A \cup I_B \qquad O_{(A,B)} = O_A \cup O_B \qquad V_{(A,B)} = V_A \cup V_B$$
$$I_{(B,C)} = I_B \cup I_C \qquad O_{(B,C)} = O_B \cup O_C \qquad V_{(B,C)} = V_B \cup V_C$$
$$I_{(A,B,C)} = I_A \cup I_B \cup I_C \; O_{(A,B,C)} = O_A \cup O_B \cup O_C \; V_{(A,B,C)} = V_A \cup V_B \cup V_C$$

and proof

$$(\sigma, i, o, \acute{\sigma}) \in [\![(A \otimes B) \otimes C]\!] \Leftrightarrow (\sigma, i, o, \acute{\sigma}) \in [\![A \otimes (B \otimes C)]\!]$$

Case "$\Rightarrow$" *Due to the hull operation we know that for all* $(\sigma, i, o, \acute{\sigma}) \in [\![(A \otimes B) \otimes C]\!]$

$$\begin{aligned}
&\exists \sigma_1, \ldots, \sigma_{n+1} \in \Sigma_{(A,B,C)}, \\
&\; i_1, \ldots, i_n \in \mathbb{H}_2(I_{(A,B,C)}), \\
&\; o_1, \ldots, o_n, \in \mathbb{H}_2(O_{(A,B,C)}) : \\
&\qquad (\bigwedge_{j=1}^{n} (\sigma_j, i_j, o_j, \sigma_{j+1}) \in [\![(A \otimes B) \otimes C]\!]) \wedge \\
&\qquad i = i_1 \frown i_2 \frown \cdots \frown i_n \wedge \\
&\qquad o = i_1 \frown o_2 \frown \cdots \frown o_n \wedge \\
&\qquad \sigma_1 = \sigma \wedge \sigma_{n+1} = \acute{\sigma}
\end{aligned}$$

Furthermore, we know that

$$(\sigma, i, o, \acute{\sigma})_{\lfloor C} \in [\![C]\!] \qquad \bigwedge_{j=1}^{n} (\sigma_j, i_j, o_j, \sigma_{j+1})_{\lfloor C} \in [\![C]\!] \qquad \wedge$$
$$(\sigma, i, o, \acute{\sigma})_{\lfloor (A,B)} \in [\![A \otimes B]\!] \qquad \bigwedge_{j=1}^{n} (\sigma_j, i_j, o_j, \sigma_{j+1})_{\lfloor (A,B)} \in [\![A \otimes B]\!]$$

and

$$(\sigma, i, o, \acute{\sigma})_{\lfloor A} \in [\![A]\!] \qquad \bigwedge_{j=1}^{n} (\sigma_j, i_j, o_j, \sigma_{j+1})_{\lfloor A} \in [\![A]\!] \qquad \wedge$$
$$(\sigma, i, o, \acute{\sigma})_{\lfloor B} \in [\![B]\!] \qquad \bigwedge_{j=1}^{n} (\sigma_j, i_j, o_j, \sigma_{j+1})_{\lfloor B} \in [\![B]\!]$$

Finally we conclude

$$\bigwedge_{j=1}^{n} (\sigma_j, i_j, o_j, \sigma_{j+1})|_{\lfloor (B,C)} \in$$
$$\{\ (\sigma, x, y, \acute{\sigma}) \mid \exists z \in \mathbb{H}(C_B \cup C_C) :$$
$$z|_{\underline{I}_{(B,C)}} = x \wedge z|_{\underline{O}_{(B,C)}} = y \quad \wedge$$
$$(\sigma, x, y, \acute{\sigma})|_{\underline{B}} \in [\![B]\!] \quad \wedge$$
$$(\sigma, x, y, \acute{\sigma})|_{\underline{C}} \in [\![C]\!]\ \}$$

This holds because the states and channels are compatible (otherwise they were not in the original behavior). With a similar argument we find that

$$(\sigma, i, o, \acute{\sigma})|_{\lfloor (B,C)} \in$$
$$\{\ (\sigma, x, y, \acute{\sigma}) \mid \exists z \in \mathbb{H}(C_B \cup C_C) :$$
$$z|_{\underline{I}_{(B,C)}} = x \wedge z|_{\underline{O}_{(B,C)}} = y \quad \wedge$$
$$(\sigma, x, y, \acute{\sigma})|_{\underline{B}} \in [\![B]\!] \quad \wedge$$
$$(\sigma, x, y, \acute{\sigma})|_{\underline{C}} \in [\![C]\!]\ \}$$

Since the set contains all single steps we conclude both $(\sigma, i, o, \acute{\sigma})|_{\lfloor (B,C)}$ *and all* $(\sigma_j, i_j, o_j, \sigma_{j+1})|_{\lfloor (B,C)}$ *are elements of* $[\![B \otimes C]\!]$

With $(\sigma, i, o, \acute{\sigma})|_{\underline{A}} \in [\![A]\!]$ *and* $\bigwedge_{j=1}^{n} (\sigma_j, i_j, o_j, \sigma_{j+1})|_{\underline{A}} \in [\![A]\!]$*, we conclude that*

$$(\sigma, i, o, \acute{\sigma}) \in [\![A \otimes (B \otimes C)]\!]$$

Case "$\Leftarrow$" *Similar to case "$\Rightarrow$"* *q.e.d.*

PROOF 14.2 (OF THEOREM 14 EQUATION (A.4)**):**
Similar to proof 14.1. The proof bases on the order of adding the behaviors being unimportant for the closure. All single step behaviors are element of the intermediate sets ensuring that the closure creates all behaviors. *q.e.d.*

Associativity and commutativity of parallel and alternative composition are a direct result from their definition that bases on logical *AND* and *OR*. Parenthesis is a means to express hierarchy in an otherwise syntactically flat service expression. The associativity allows restructuring the hierarchy within some sub-expression of equal operators. The law of distributivity allows a more general restructuring.

THEOREM 15 (DISTRIBUTIVITY FOR PARALLEL AND ALTERNATIVE COMPOSITION)

$$A \otimes (B \oplus C) = (A \otimes B) \oplus (A \otimes C) = (B \oplus C) \otimes A \tag{A.5}$$
$$A \oplus (B \otimes C) = (A \oplus B) \otimes (A \oplus C) = (B \otimes C) \oplus A \tag{A.6}$$

PROOF 15.1 (OF THEOREM 15 EQUATION (A.5)):

"⇒" *First we observe that* $(\sigma, i, o, \acute{\sigma})$ *can be decomposed in arbitrary fragments since* $(A \otimes (B \oplus C))$ *is CWR. We refer to those fragments as* $(\sigma_i, i_i, o_i, \sigma_{i+1})$.

$$(\sigma, i, o, \acute{\sigma}) \in A \otimes (B \oplus C) \Rightarrow$$
$$(\sigma, i, o, \acute{\sigma})_{\lfloor A} \in A \wedge (\sigma, i, o, \acute{\sigma})_{\lfloor (B,C)} \in (B \oplus C)$$

Furthermore, we know that

$$\begin{aligned}
&(\sigma, i, o, \acute{\sigma})_{\lfloor B} \in [\![B]\!] \vee \\
&(\sigma, i, o, \acute{\sigma})_{\lfloor C} \in [\![C]\!] \vee \\
&\exists\, \sigma_1, \ldots, \sigma_n \in \Sigma_{(A,B,C)}, \\
&\quad i_1, \ldots, i_{n-1} \in \mathbb{H}(I_{(A,B,C)}), \\
&\quad o_1, \ldots, o_{n-1}, \in \mathbb{H}(O_{(A,B,C)}) : \\
&\qquad \bigwedge_{j=1}^{n} ((\sigma_j, i_j, o_j, \sigma_{j+1})_{\lfloor B} \in [\![B]\!] \vee (\sigma_j, i_j, o_j, \sigma_{j+1})_{\lfloor C} \in [\![C]\!]) \wedge \\
&\qquad i = i_1 \frown i_2 \frown \cdots \frown i_n \wedge \\
&\qquad o = i_1 \frown o_2 \frown \cdots \frown o_n \wedge \\
&\qquad \sigma_1 = \sigma \wedge \sigma_{n+1} = \acute{\sigma}
\end{aligned} \tag{A.7}$$

Informally spoken, either the behavior is element of B (subsequently case 1), of C (subsequently case 2), or is a combination of behaviors of B and C (subsequently case 3).

Case 1 and 2 *We can immediately conclude that*

$$(\sigma, i, o, \acute{\sigma})_{\lfloor (A,B)} \in [\![A \otimes B]\!] \vee$$
$$(\sigma, i, o, \acute{\sigma})_{\lfloor (A,C)} \in [\![A \otimes C]\!]$$

and we conclude that

$$(\sigma, i, o, \acute{\sigma}) \in [\![(A \otimes B) \oplus (A \otimes C)]\!]$$

Case 3 *Because* $(A \otimes (B \oplus C))$ *is CWR we know that there are* $(\sigma_i, i_i, o_o, \sigma_{i+1})$ *such that*

$$(\sigma_i, i_i, o_i, \sigma_{i+1})_{\lfloor A} \in A \wedge (\sigma_i, i_i, o_i, \sigma_{i+1})_{\lfloor (B,C)} \in (B \oplus C)$$

we are now free to choose the $(\sigma_i, i_i, o_i, \sigma_{i+1})$ *such that they correspond to the* $(\sigma_j, i_j, o_j, \sigma_{j+1})$ *in Expression* (A.7)*. Therefore we know that the*

$$((\sigma_j, i_j, o_j, \sigma_{j+1}) \in (A \otimes B)) \vee ((\sigma_j, i_j, o_j, \sigma_{j+1}) \in (A \otimes C))$$

With the transitive hull of the $\mathcal{CL}$ *operation in the alternative composition we conclude that* $(\sigma, i, o, \acute{\sigma})$ *is in* $[\![(A \otimes B) \oplus (A \otimes C)]\!]$ *as well.*

"$\Leftarrow$" *Again we start with the observation that either the behaviors are completely contained in* $A \otimes B$ *or* $A \otimes C$ *or that there are fragments which satisfy this property:*

$$\begin{aligned}
&(\sigma, i, o, \acute{\sigma}) \in (A \otimes B) \oplus (A \otimes C) \Rightarrow \\
&(\sigma, i, o, \acute{\sigma})_{\lfloor A,B} \in [\![A \otimes B]\!] \vee \\
&(\sigma, i, o, \acute{\sigma})_{\lfloor A,C} \in [\![A \otimes C]\!] \vee \\
\exists \quad &\sigma_1, \ldots, \sigma_n \in \Sigma_{(A,B,C)}, \\
&i_1, \ldots, i_{n-1} \in \mathbb{H}(I_{(A,B,C)}), \\
&o_1, \ldots, o_{n-1}, \in \mathbb{H}(O_{(A,B,C)}) : \\
&\quad \bigwedge_{j=1} ((\sigma_j, i_j, o_j, \sigma_{j+1})_{\lfloor A,B} \in [\![A \otimes B]\!] \vee (\sigma_j, i_j, o_j, \sigma_{j+1})_{\lfloor A,C} \in [\![A \otimes C]\!]) \wedge \\
&\quad i = i_1 \frown i_2 \frown \cdots \frown i_n \wedge \\
&\quad o = i_1 \frown o_2 \frown \cdots \frown o_n \wedge \\
&\quad \sigma_1 = \sigma \wedge \sigma_{n+1} = \acute{\sigma}
\end{aligned}$$

All fragments need to be valid behaviors for A *and* $CWR(A)$*. Therefore,*

$$(\sigma, i, o, \acute{\sigma})_{\lfloor A} \in [\![A]\!],$$

otherwise they were no valid behaviors of $A \otimes B$ *or* $A \otimes C$*. Furthermore, we know that the fragments either need to be valid behaviors for* B *or* C*. Due to the closure of alternative composition we conclude that*

$$(\sigma, i, o, \acute{\sigma})_{\lfloor B,C} \in [\![B \oplus C]\!],$$

Finally, we know that $CWR(A)$*,* $CWR(B)$ *and* $CWR(C)$*. We conclude that all states, inputs and outputs in every step of* $(\sigma, i, o, \acute{\sigma})$ *are compatible with* A*,* B *and* C*. Therefore, the hull operation in the parallel composition does not remove* $(\sigma, i, o, \acute{\sigma})$ *from the set of valid behaviors and we conclude that*

$$(\sigma, i, o, \acute{\sigma}) \in [\![A \otimes (B \oplus C)]\!] \qquad \text{q.e.d.}$$

PROOF 15.2 (OF THEOREM 15 EQUATION (A.6)):
To show:

$$A \oplus (B \otimes C) = (A \oplus B) \otimes (A \oplus C) = (B \otimes B) \oplus A$$

The proof is analogous to proof 15.1. *q.e.d.*

A number of other relevant properties (including those of the unary operators) exist that allow creating an analytical normal form as described in Section 6.2.2.

THEOREM 16 (NEUTRALITY OF TRUE AND FALSE)

$$TRUE \otimes A = A \qquad \textit{TRUE is neutral to } \otimes \textit{ (see Proof 16.1)} \qquad \text{(A.8)}$$
$$FALSE \oplus A = A \qquad \textit{FALSE is neutral to } \oplus \textit{ (see Proof 16.1)} \qquad \text{(A.9)}$$

PROOF 16.1 (OF THEOREM 16 EQUATION (A.8)):
$TRUE \otimes A = A$

$$\begin{aligned}
&(\sigma, x, y, \acute{\sigma}) \in (TRUE \otimes A) && \Leftrightarrow \\
&(\sigma, x, y, \acute{\sigma}) \in TRUE \wedge (\sigma, x, y, \acute{\sigma}) \in A && \Leftrightarrow \\
&(\sigma, x, y, \acute{\sigma}) \in A
\end{aligned}$$

The first step is valid because TRUE contains every behavior and the hull operation does not remove any behavior from A. With the commutativity of parallel compositions $A \otimes TRUE = A$ holds as well. *q.e.d.*

PROOF 16.2 (OF THEOREM 16 EQUATION (A.9)):
$FALSE \oplus A = A$

$$\begin{aligned}
&(\sigma, x, y, \acute{\sigma}) \in (FALSE \oplus A) && \Leftrightarrow \\
&(\sigma, x, y, \acute{\sigma}) \in FALSE \vee (\sigma, x, y, \acute{\sigma}) \in A && \Leftrightarrow \\
&(\sigma, x, y, \acute{\sigma}) \in \{\} \vee (\sigma, x, y, \acute{\sigma}) \in A && \Leftrightarrow \\
&(\sigma, x, y, \acute{\sigma}) \in A
\end{aligned}$$

The first operation is valid because FALSE does not introduce a behavior. Hence, the closure operation does not extend the behaviors of A.
With the commutativity of alternative compositions $A \oplus FALSE = A$ holds as well. *q.e.d.*

TRUE and *FALSE* are neutral services for the two basic binary operators. Most notably is *TRUE* as it occurs as a mode-behavior in unbundling (cf. Section 5.4).

THEOREM 17 (IDEMPOTENCE OF PARALLEL AND ALTERNATIVE COMPOSITION)
Without proof:

$$S_1 \otimes S_1 = S_1 \qquad \textit{Idempotence for } \otimes \qquad \text{(A.10)}$$

$$S_1 \oplus S_1 = S_1 \qquad \textit{Idempotence for } \oplus \qquad \text{(A.11)}$$

□

A.3.2. Transformations for mode transition systems

The validity of the distribution of a parallel composition of an arbitrary service B over a mode transition system A given as a specification (M_A, δ_A, Φ_A) strongly depends on the consistency with resumption of B. The idea is that a parallel composed service B always runs in parallel with any of the mode-behaviors of A. However, if we intend to distribute a service over a mode transition system we have to accept that the distributed service (here B) has to allow interruptions and resumptions according to the mode changes. Furthermore, the service B must be able to immediately resume its behavior thus to simulate a behavior as if it was never interrupted. This ensures an equivalent behavior to the original service B that never is affected by a mode change.

THEOREM 18 (DISTRIBUTION OF PARALLEL COMPOSITION OVER MTS)
Let $A = (M_A, \delta_A, \Phi_A)$ be a mode transition system and B an arbitrary service.

$$(M_A, \delta_A, \Phi_A) \otimes B = A'$$

such that

- $A' = (M_{A'}, \delta_{A'}, \Phi_{A'})$ *with*
- $M_{A'} = M_A$
- $\delta_{A'} = \delta_A$
- $\forall\, m \in M_{A'} : \Phi_{A'}(m) = \Phi_A(m) \otimes B$

PROOF 18.1 (OF THEOREM 18):
The proof idea bases on the observation that mode transition systems are similar to alternative composition but do not paste together all possible behaviors of the operands but only those whose inputs satisfy the transition conditions respectively.

"⇒": $(\sigma, i, o, \acute{\sigma}) \in [\![A \otimes B]\!] \Rightarrow (\sigma, i, o, \acute{\sigma}) \in [\![A']\!]$
We observe that

$$(\sigma, i, o, \acute{\sigma}) \in [\![A \otimes B]\!] \Rightarrow (\sigma, i, o, \acute{\sigma}) \in [\![A]\!] \wedge (\sigma, i, o, \acute{\sigma}) \in [\![B]\!]$$

Case 1: *If A does not execute a mode transition based on the inputs i, we know that*

$$(\sigma, i, o, \acute{\sigma}) \in [\![A]\!] \Rightarrow (\sigma, i, o, \acute{\sigma})_{\lfloor \Phi_A(\sigma(PC_A))} \in [\![\Phi_A(\sigma(PC_A))]\!]$$

From that we immediately conclude that

$$(\sigma, i, o, \acute{\sigma})_{\lfloor \Phi_A(\sigma(PC_A)),B} \in [\![\Phi_A(\sigma(PC_A)) \otimes B]\!]$$

and that the property holds.

Case 2: *If A executes one or more mode transitions*

$$\begin{aligned}
\exists \quad & \sigma_1, \ldots, \sigma_n \in \Sigma, \\
& i_1, \ldots, i_{n-1} \in \mathbb{H}(I), \\
& o_1, \ldots, o_{n-1} \in \mathbb{H}(O), \\
& (m_1, (c_1, \varsigma), m_2), \ldots, (m_{j-3}, (c_{j-3}, \varsigma), m_{j-2}) \in \delta_A \\
& \quad \bigwedge_{j=1}^{n} ((\sigma_j, i_j, o_j, \sigma_{j+1})_{\lfloor \Phi(\sigma_j(PC_A))} \in [\![\Phi(\sigma_j(PC_A))]\!] \wedge \Theta(\sigma_j, c_{j-1}, i_j, o_j)) \\
& \quad i = i_1 \frown i_2 \frown \cdots \frown i_n \wedge \\
& \quad o = i_1 \frown o_2 \frown \cdots \frown o_n \wedge \\
& \quad \sigma_1 = \sigma \wedge \sigma_{n+1} = \acute{\sigma}
\end{aligned}$$

Informally spoken, there is a decomposition of $(\sigma, i, o, \acute{\sigma})$ into a sequence of behaviors that correspond to a sequence of mode changes of the mode transition system. By CWR(B) we know that there are sequences in $[\![B]\!]$ which correspond to this decomposition. Furthermore, $(\sigma, i, o, \acute{\sigma}) \in [\![A \otimes B]\!]$ implies that these parts are compatible with respect to their states, inputs and outputs. For each of these parts Case 1 applies i.e.:

$$\begin{aligned}
& (\sigma_j, i_j, o_j, \sigma_{j+1})_{\lfloor A} \in [\![A]\!] \wedge (\sigma_j, i_j, o_j, \sigma_{j+1})_{\lfloor B} \in [\![B]\!] \Rightarrow \\
& (\sigma, i, o, \acute{\sigma})_{\lfloor \Phi_A(\sigma(PC_A)),B} \in [\![\Phi_A(\sigma(PC_A)) \otimes B]\!]
\end{aligned}$$

The mode transition system definition is recursive and includes all behaviors of any length that satisfy the mode transition system definition. B does not influence the sequence of mode changes (because the inputs and outputs correspond) and the final and initial states of the parts match we conclude that their sequence is a valid behavior for A′:

$$(\sigma, i, o, \acute{\sigma}) \in [\![A']\!]$$

"$\Leftarrow$" $(\sigma, i, o, \acute{\sigma}) \in [\![A']\!] \Rightarrow (\sigma, i, o, \acute{\sigma}) \in [\![A \otimes B]\!]$

Case 1: *If A' executes no mode transition, obviously,*

$$(\sigma, i, o, \acute{\sigma})_{\lfloor B} \in [\![B]\!] \wedge (\sigma, i, o, \acute{\sigma})_{\lfloor \Phi(\sigma(PC_A))} \in [\![\Phi(\sigma(PC_{A'}))]\!]$$

where $\Phi(\sigma(PC_{A'})) = \Phi(\sigma(PC_A) \otimes B)$. Since B and $\Phi(\sigma(PC_A)$ are CWR, the hull operation of the parallel composition will not remove the behavior. This immediately implies that $(\sigma, i, o, \acute{\sigma}) \in [\![A \otimes B]\!]$

Case 2: *If A' executes one or more mode transitions,*

$$\begin{aligned}
\exists \quad & \sigma_1, \ldots, \sigma_n \in \Sigma, \\
& i_1, \ldots, i_{n-1} \in \mathbb{H}(I), \\
& o_1, \ldots, o_{n-1} \in \mathbb{H}(O), \\
& (m_1, (c_1, \varsigma), m_2), \ldots, (m_{j-3}, (c_{j-3}, \varsigma), m_{j-2}) \in \delta_{A'} \\
& \quad \bigwedge_{j=1}^{n} ((\sigma_j, i_j, o_j, \sigma_{j+1})_{\lfloor \Phi(\sigma_j(PC_{A'}))} \in [\![\Phi(\sigma_j(PC_{A'}))]\!] \wedge \Theta(\sigma_j, c_{j-1}, i_j, o_j)) \\
& \quad i = i_1 \frown i_2 \frown \cdots \frown i_n \wedge \\
& \quad o = i_1 \frown o_2 \frown \cdots \frown o_n \wedge \\
& \quad \sigma_1 = \sigma \wedge \sigma_{n+1} = \acute{\sigma}
\end{aligned}$$

Informally spoken, the behavior can be decomposed into parts that comply with the mode-behaviors according to the sequence of the mode changes. According to the definition of A' the mode-behaviors always look like

$$\Phi(\sigma_j(PC_A)) \otimes B$$

That in turn implies that the parts must be valid for B and for $\Phi(\sigma_j(PC_A))$. Because the mode transition system definition requires that the initial and final states match in a sequence of mode-behaviors the validity also applies for $(\sigma_j, i_j, o_j, \sigma_{j+1})_{\lfloor B}$. Since B is CWR we may conclude that

$$(\sigma, i, o, \acute{\sigma})_{\lfloor B} \in [\![B]\!]$$

with the same argument we see that

$$(\sigma, i, o, \acute{\sigma})_{\lfloor A} \in [\![A]\!]$$

Because CWR(A) and CWR(B) all decompositions of $(\sigma, i, o, \acute{\sigma})$ are valid behaviors for A and B as well and the hull operation of the parallel composition will not remove $(\sigma, i, o, \acute{\sigma})$ from the set of valid behaviors. Therefore,

$$(\sigma, i, o, \acute{\sigma}) \in [\![A \otimes B]\!]$$ q.e.d.

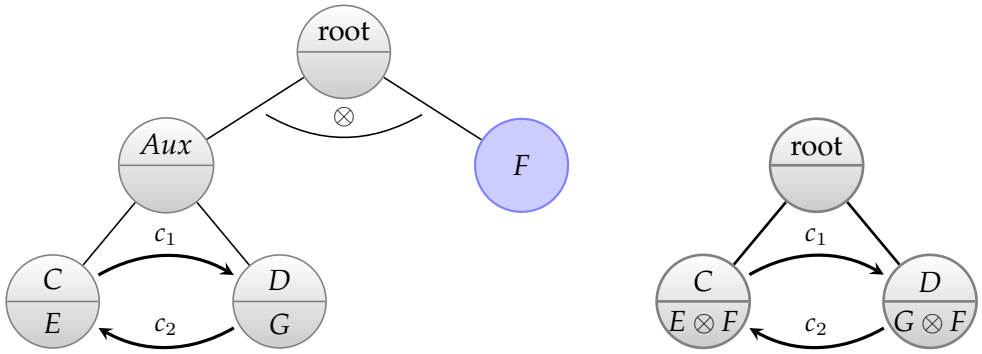


Figure A.6.: Transformation of a simple mode transition system that is parallel composed with some service F in to a new mode transition system with F integrated in the mode-behaviors

Figure A.6 illustrates the distribution for a mode transition system with two modes and mode-behaviors E and G and a parallel composed service F.

Note that we required nothing special for service F. Especially there is no assumption about the structure of F. It can be any service expression including a mode transition system. The transformation distributes the mode transition system of F over the mode-behaviors of A. Of course it is possible to iteratively apply this transformation for the mode-behaviors. The result is a nested mode transition system.

A.4. Refinement and Equivalence

Equivalence and refinement are two important relations for algebraic structures that allow comparing instances of a domain. MARLIN is an algebraic structure and equivalence and refinement compare the behavior of services. While a simple equivalence is just a symmetric, reflexive and transitive relation, a congruence additionally requires that functions application maintains the equivalence. In the context of service based software engineering this corresponds to compositionality of services.

The goal of any specification is to describe the properties of a system under development. Different levels of abstraction of the specifications add information like restrictions of the architecture or even detailed information about the deployment platform.

Adding information is to confine the solution space. While doing this the original functionality must remain the same. Therefore, any of the artifacts have to be related in an adequate manner i.e. each artifact has to maintain the properties of its predecessor. In software engineering this is known as refinement (e.g., [Dijkstra, 1968; Dahl et al., 1972; Hoare, 1997]). Refinement is a generalization of an equivalence and only claims for transitivity and reflexivity. Compositionality is applicable to refinement relations as well and is an important property in software engineering if it comes to relating different artifacts. In the sequel we discuss the notions of equivalence and refinement in the context of MARLIN

A.4.1. Property refinement

Refinement relates the behavior of a more detailed specification (or even implementation) with a preceding, more abstract specification. Usually, each behavior of the refined specification must be a valid behavior of the more abstract one. Therefore, refinement is a logical implication between the sets of behaviors: given two specifications A and C as logical formulas, C is a refinement of A written as $C \rightsquigarrow A$ (C refines A) iff $C \Rightarrow A$. If A and C are directly given as sets, the relation is $C \rightsquigarrow A$ iff $C \subseteq A$ [Broy and Stølen, 2001; Hoare, 1997].

In the context of component based specifications like presented by Broy [Broy and Stølen, 2001] this notion of refinement is sufficient. Components are total functions. Hence, each refinement of a component is a component as well but possibly discards some non-determinism. However, in other settings we need different definitions to I) capture the idea of refinement in the face of partiality and II) to guaranty compositionality (for an introduction to the relation between semantics, semantic domains and compositionality see, e.g., [Olderog, 1985] and [Olderog, 1986]).

In the context of service based development we must reconsider the use of refinement. The services define no obligation for a structure (in contrast to components). Therefore, a modular refinement that allows replacing any service by an equivalent one is less important.

According to Hoare [Hoare, 1997] the most general but at the same time most useless specification is *TRUE*. With the statement that a specification describes all acceptable implementations, we see that *TRUE* accepts any implementation. Even those without reasonable behavior. In contrast, *FALSE* accepts no implementation. In practice a good specification defines something in-between. It is as liberal as possible to allow many technical solutions and as restrictive as possible to avoid unwanted behavior.

Initial requirements that are conflicting and incomplete are equal to *FALSE*. It is the use of a service based specification to enable the analysis for such contradictions to address them.

Due to the partial nature of services, service refinement must consider changing the domains. Therefore we need to make a decision what to accept as a refinement and which properties to preserve by refinement. If we allow for a decrease of the domain the trivial service *FALSE* is always a proper refinement of every service. This is undesirable. Increasing the domain involves two challenges: I) a naive extension of the domain may result in non-causal behavior and II) the extension of a domain may affect the validity of properties. We shortly discuss both aspects before we introduce the notion of refinement in the context of MARLIN.

Violation of causality Broy defines the closure of a service as the least refinement of a service that is a component [Broy, 2005] but excludes the naive closure. A naive closure assigns arbitrary behavior to each input that is not in the domain:

$$F_{naiveclosure}.(\sigma, x') = \{(y, \acute{\sigma}) \mid (\sigma, x) \in dom(F) \Rightarrow (\sigma, x, y, \acute{\sigma}) \in [\![F]\!]\}$$

Any partial function's domain may contain input-histories whose prefix is equal

to an input history that is not in the domain i.e. (adapted to the semantics of MARLIN)

$$\exists \sigma, \sigma_n, x, x', t : (\sigma, x) \in dom(F) \wedge (\sigma, x') \notin dom(F) \wedge (x)_{\downarrow t} = (x')_{\downarrow t}$$

For such prefix it is unclear if the inputs continue in a way that matches to the input stream in the domain or not. If it is not, the naive closure allows arbitrary behavior. On the other hand, if it is in the domain only the behavior according to F is allowed. As a consequence, $F_{naiveclosure}$ would have to "know" how the inputs continue to know whether the input is in the domain or not. This contradicts causality. This issue applies even to small extensions of the domain.

Violation of properties Following the lines of branching time temporal logics (CTL*) we may observe basically two types of properties. I) there exists some I/O-history with a certain property and II) for all I/O-histories a certain property holds.

Refinements that base on removing non-determinism in general preserve no properties of the first kind. A small counter example illustrates this. A service F with a non-deterministic choice for an output satisfies such properties even if only one of the choices satisfies the property. A refinement that allows removing non-determinism allows removing exactly the one I/O history that satisfies the property. As a result, the refined service does not satisfy the property any more. However, these properties are less important for us. Fairness requirements can be expressed with this property but in general we are more interested in properties that hold on all paths.

In general refinements preserve properties of the second kind only if the domain is not expanded. Once a property has been proven for all I/O-histories of a service a refinement only reducing non-determinism does not affect the property. However by allowing to expand the domain by a refinement the refining service may contain additional I/O-histories that do not satisfy the property any more. Rather than

$$F_1 \rightsquigarrow F_2 \Rightarrow$$
$$((\forall(\sigma, x, y, \acute{\sigma}) \in [\![F_1]\!] : (\sigma, x, y, \acute{\sigma}) \models p) \Rightarrow (\forall(\sigma, x, y, \acute{\sigma}) \in [\![F_2]\!] : (\sigma, x, y, \acute{\sigma}) \models p))$$

we only may conclude a restricted property from the refinement relation:

$$F_1 \rightsquigarrow F_2 \Rightarrow$$
$$(\forall(\sigma, x, y, \acute{\sigma}) \in [\![F_1]\!] : (\sigma, x, y, \acute{\sigma}) \in F_2 \Rightarrow (\sigma, x, y, \acute{\sigma}) \models p$$

In [Broy, 2005] expanding a domain reduces partiality and transfers a service it into a component. MARLIN offers dedicated means for this purpose by alternative composition. Therefore, we may renounce the expansion of the domain in favor of a more comprehensive notion of property preservation. This has the benefit that each change of a specification towards a less partial one is a composition of services that needs to be

subject to verification. Alternative composition preserves properties on all paths in the composed service if the property holds on all paths of the sub-services.

DEFINITION 55 (SERVICE REFINEMENT):
Let $F_1 \in \mathbb{F}(I_1 \rhd O_1, V_1)$ and $F_2 \in \mathbb{F}(I_2 \rhd O_2, V_2)$ with $I_1 \subseteq I_2$, $O_1 \subseteq O_2$ and $V_1 \subseteq V_2$. We call F_2 a refinement of F_1 iff

$$dom(F_1) \subseteq \{(x_{\lfloor I_1}, \sigma_{\lfloor v_1}) \mid (x, \sigma) \in dom(F_2)\} \quad \wedge \tag{A.12}$$

$$\{(y, \acute{\sigma}) \mid (y, \acute{\sigma}) = F_1.(x, \sigma)\} \subseteq \{(y_{\lfloor I_1}, \acute{\sigma}_{\lfloor V_1}) \mid (y, \acute{\sigma}) = F_2.(x, \sigma)\} \quad \wedge \tag{A.13}$$

$$CWI(F_2) \tag{A.14}$$

□

Note that we explicitly allow the domain to decrease and the sets of channels to increase. Both does not affect our demand for preserving properties holding for all I/O-streams. Furthermore we explicitly require a refinement to be consistent with interruption. The operators of MARLIN maintain the property according to Theorem 12.

Refinement allows relating consecutive steps of the development. An equivalence allows relating services within one level of abstraction. Nevertheless, equivalence is just a special case of refinement.

DEFINITION 56 (EQUIVALENCE):
Let $F_1 \in \mathbb{F}(I_1 \rhd O_1, V_1)$ and $F_2 \in \mathbb{F}(I_2 \rhd O_2, V_2)$. We say that F_1 and F_2 are equivalent, written as

$$F_1 = F_2 \Leftrightarrow [\![F_1]\!] = [\![F_2]\!] \Leftrightarrow$$
$$\{(\sigma, x, y, \acute{\sigma}) \mid (\sigma, x, y, \acute{\sigma}) \in [\![F_1]\!]\} = \{(\sigma, x, y, \acute{\sigma}) \mid (\sigma, x, y, \acute{\sigma}) \in [\![F_2]\!]\}$$

□

Two services are equivalent if the sets of their behaviors are equivalent. Of course this requires that their channel sets and types are equal as well. As with any equivalence the equivalence of services is transitive, reflexive and symmetric (without proof, clear from set theory).

A.4.2. Compositionality

Compositionality allows inferring the semantics of any composed behavior from the behaviors of its constituents.

- a service F_2 that is a service refinement of a service F_1 may replace F_1 in a service expression such that the resulting expression is a refinement of the original expression.
- equivalent services F_1 and F_2 are exchangeable in any service expression without changing the meaning of that expression.

We already argued that service refinement is a generalization of service equivalence. Therefore we discuss this property only with respect to service refinement.

DEFINITION 57 (COMPOSITIONALITY):
Let $*$ be any arbitrary operator of a specification language. We call this language compositional iff:

$$F_1 \rightsquigarrow F_1' \Rightarrow \begin{cases} (F_1 * F_2) \rightsquigarrow (F_1' * F_2) & \text{if } * \text{ is a binary operator and} \\ *(F_1, \ldots, F_n) \rightsquigarrow *(F_1', \ldots, F_n) & \text{if } * \text{ is n-ary} \end{cases} \tag{A.15}$$

For mode transition systems we may regard this as a n-ary operator. The idea of compositionality applies in a straight forward manner: if one of the mode-behaviors is refined this implies that the whole mode transition systems is refined. □

THEOREM 19 (COMPOSITIONALITY OF MARLIN)
Alternative-, parallel-, and conditional composition in MARLIN *are compositional.* □

Without formal proof because refinement and compositionality are no major aspect of the methodology of MARLIN. They are more important for the subsequent steps where, e.g., the definitions of Broy et al. [Broy and Stølen, 2001] for refinement and compositionality apply.

Informally spoken, a refined service is more restricted. In a parallel composition these restrictions carry over to common I/O-histories i.e. inputs that are removed from the domain of the restricted service are also removed from the compound service. In alternative and conditional composition the refined service does not contribute the removed I/O histories to the set of behaviors of the compound service. Always the domain may decrease, the number of channels may increase and the set of behaviors of the refined compound service is a subset of the behaviors of the original compound service.

Appendix B

Case study

Throughout the thesis we use a running example. This running example bases on a case-study from the automotive domain. The case study originally originates from a real industrial case study and was part of a bilateral collaboration. To account for the original property rights of the case study we made significant changes:

- the case study now contains common functionality for locking doors
- conditions for the activation of functions are different
- message characteristics are altered and simplified to downsize the case study

Subsequently we provide coherent information about the case-study. The specification of the Car Entry System `CESys` summarizes the interface of the case study's system as according to Section 1.5. In the sequel we omit the types of the channels and variables for reasons of brevity.

We repeat the abbreviations:

Context There are three context aspects that the system takes into account

- `KPos` is a valid key at a certain position
  - `dd` = driver door
  - `pd` = passenger door
  - `tr` = trunk
  - `in` = inside
  - `away` = away
- `V` is the velocity
  - `hi` = greater than zero
  - `lo` = equal zero
- `UBat` is the battery
  - `hi` = high capacity
  - `lo` = low capacity

[CESys]

in	$V : \{lo, hi\}, UBAT : \{hi, low\}, KPos : \{dd, pd, tr, in, away\}, ER : \{on, off\},$ $TA : \{t, op, cl\}, DA : \{t, in\}, PA : \{i, in\}, DL : \{l\}$
out	$EC : \{on, off\}, DC : \{l, u\}, PC : \{l, u\}, TC : \{l, u\}, TD : \{op, cl\}$
var	$DDSt : \{cl, ou, cu\}, PDSt : \{cl, ou, cu\}, TDSt : \{cl, ou, cu\}$

Inputs The system accepts five different inputs

- `ER` represents a user's wish to start or stop the engine
- `DA`, `PA` and `TA` are the driver door, passenger door and trunk door triggers respectively
  - `t` = outside triggered
  - `op` = executing an opening gesture
  - `cl` = executing a closing gesture
- `DL` represents a user wish to lock all doors.

Outputs The system has five output channels to react to inputs

- `EC` is the actual command to the engine whether to start or turn off.
- `DC`, `PC` and `TC` are the driver door, passenger door and trunk door commands
  - `l` = lock
  - `u` = unlock
- `TD` is the trunk door engine
  - `op` is the signal for the trunk door engine to physically open
  - `cl` is the signal for the trunk door engine to physically close

Variables `DDSt`, `PDSt`,`TDSt` are state variables indicating the state of the doors

- `cl` closed and locked
- `cu` closed but unlocked
- `ou` open and unlocked

To keep the figures small and simple we use the following abbreviations for states in the figures:

- $\texttt{dl} \stackrel{def}{=} \texttt{DDSt} = \texttt{cl}$
- $\texttt{du} \stackrel{def}{=} \texttt{DDSt} = \texttt{cu}$
- $\texttt{do} \stackrel{def}{=} \texttt{DDSt} = \texttt{ou}$

for the driver door and in the same way for the passenger door (`pl`, `pu`, `po`) and the trunk door (`tl`, `tu`, `to`)

In the case study we allow no locking of doors that are open. Therefore, we need no state for open and locked. Since we use nodes as control states that represent an equivalence class of a set of data states we subsequently use control state names like dl, pl instead of $\texttt{DDSt} = \texttt{cl} \wedge \texttt{PDSt} = \texttt{cl}$

B.1. Enhanced Use-Cases and Requirements

We start with the (initial) requirements to the Car Entry System. The set of requirements easily can be identified as incomplete. Many of the interactions we expect with a real car entry system are missing or are unspecified. Note that this underspecification here is intended to keep the case study simple. However, defects in the requirements like unintended underspecification or contradictions are likely to exist in the initial set of requirements.

We are not interested in methodological aspects of the requirements engineering. We present a possible formalization of the requirements already. Sometimes the requirements cover aspects of different parts of the system (like different doors). We then provide formalizations according to the requirement but occasionally feel free to further decompose the formalization.

UC1

Short description: *Doors are locked while driving*

Context: $V = hi$

Requirement: *While driving the doors are locked*

[Drive]

in
out DC, PC, TC
var $DDSt, PDSt, TDSt$

$F_{1,D} \otimes F_{1,P} \otimes F_{1,T}$

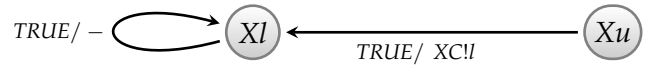
We present the sub-specifications as templates such that each of the specifications is an instance that results from substitution of the respective door. Note that simply an abbreviation.

$[F_{1,X \text{with } X \in \{d,p,t\}}]$

in

out XC

var $XDSt$



UC2

Short description: *Trunk opens on gesture*

Context:

- $V = lo$
- $UBat = hi$
- $KPos = tr$
- *System is not in protection mode*

Preconditions:

- *Trunk is closed*

Actions:

1. *Person makes opening gesture*
2. *Trunk is opened*

Postconditions:

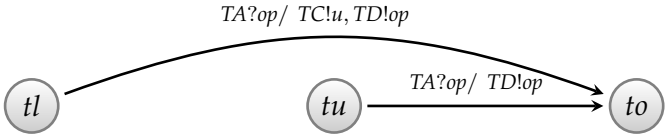
- *Trunk is unlocked*
- *Trunk is open*

$[F_2]$

in TA

out TC, TD

var $TDSt$



UC3

Short description: *Trunk closes on gesture*

Context:

- $V = lo$
- $UBat = hi$
- $Key = tr$
- *System is not in protection mode*

Preconditions:

- *Trunk is open*

Actions:

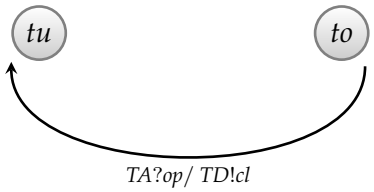
1. *Person makes closing gesture*
2. *Trunk is closed*

Postconditions:

- *Trunk is closed*

$[F_3]$

in	*TA*
out	*TD*
var	*TDSt*



UC4

Short description: *Trunk unlocks on touch*

Context:

- $V = lo$
- $KPos = tr$

Preconditions:

- *Trunk is close*

Actions:

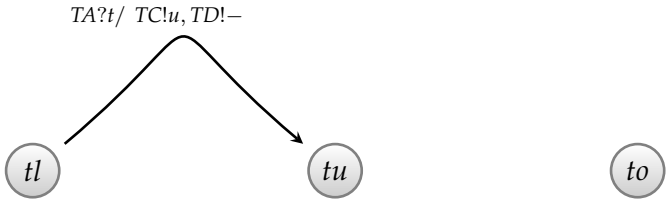
1. *Person touches trunk handle*
2. *Trunk is unlocked*
3. *Trunk stays closed for safety reasons*

Postconditions:

- *Trunk is unlocked*
- *Trunk is still closed*

$[\mathrm{F}_4]$

in	TA
out	TC, TD
var	$TDSt$



UC5

Short description: *Front doors lock on touch individually if key is at a front door position*

Context:

- $V = lo$
- $KPos \in \{dd, pd\}$

Preconditions:

- *Door is unlocked and closed*

Actions:

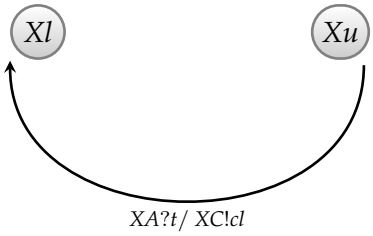
1. *Person touches handle*
2. *Door is locked*

Postconditions:

- *Door is locked*

$[\mathrm{F}_{5,\mathrm{X\ with\ X}\in\{\mathrm{D,P}\}}]$

in	XA
out	XC
var	$XDSt$



UC6

Short description: *Trunk door locks on touch if key is at trunk door*

Context:

- $V = lo$
- $KPos = tr$

Preconditions:

- *Door is unlocked and closed*

Actions:

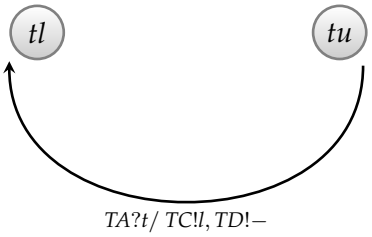
1. *Person touches handle*
2. *Door is locked*

Postconditions:

- *Door is locked*

$[\mathrm{F}_6]$

in	TA
out	TD, TC
var	$TDSt$



UC7

Short description: *No effect of gesture if battery is low or in protection mode*

Context:

- $V = lo$
- $UBat = lo \vee prt$
- $KPos = tr$

Preconditions:

- *Trunk is close*

Actions:

1. *Person makes opening gesture*
2. *No effect is observable*

Postconditions:

- *Trunk is still closed*

[F_7]

in	TA
out	TC, TD
var	$TDSt$

UC8

Short description: *All doors unlock if driver door is touched and was locked. If the driver door is unlocked or no trigger is pressed, the driver door states do not change.*

Context:

- $V = lo$
- $KPos = \{dd, pd\}$

Preconditions:

- *Driver door is locked*

Actions:

1. *Person touches driver door handle*
2. *All doors are unlocked*

Postconditions:

- *All doors are unlocked*

[$F_{8,D}$]

in	DA
out	DC
var	$DDSt$

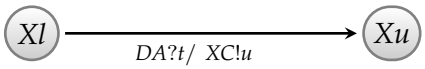
$F_{8,Da} \oplus F_{8,Db} \oplus F_{8,Dc} \oplus F_{8,Dd}$

[$F_{8,Da}$]

in *DA*
out *DC*
var *DDSt*

[$F_{8,Xb\ with\ X \in D,P,T}$]

in *DA*
out *XC*
var *XDSt*



[$F_{8,Dc}$]

in *DA*
out *DC*
var *DDSt*

[$F_{8,Dd}$]

in *DA*
out *DC*
var *DDSt*

UC9

Short description: *All doors lock if the "lock doors" signal is received.*

Context:

- $V = lo$

Preconditions:

- *Driver door is unlocked*

Actions:

1. *Person touches driver door handle*
2. *All doors are locked*

Postconditions:

- *All doors are locked*

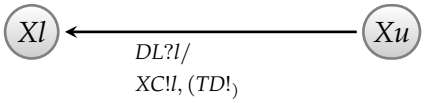
$[\mathrm{F}_9]$

in	DA
out	DC, PC, TC, TD
var	$DDSt, PDSt, TDSt$

$F_{9,D} \otimes F_{9,P} \otimes F_{9,T}$

$[\mathrm{F}_{9,\mathrm{X\ with\ X\in D,P,T}}]$

in	DA
out	$XC, (TD)$
var	$XDSt$



UC10

Short description: *Only passenger door unlocks if passenger door is touched*

Context:

- $Velocity = 0$
- $KPos \in \{dd, pd\}$

Preconditions:

- *Passenger door is locked*

Actions:

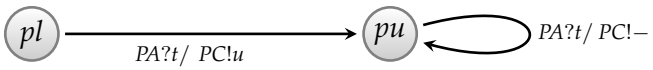
1. *Person touches PD-handle*
2. *PD is unlocked*

Postconditions:

- *PD is unlocked*

[$F_{10,P}$]

in *PA*
out *PC*
var *PDSt*



[$F_{10,X}$ with $X \in D, T$]

in *XA*
out *XC*
var *XDSt*

UC11

Short description: *All doors lock if no key is present*

Context: $KPos = away$

Requirement *If no Key is in range (last key leaves range) all doors become locked*

[F_{11}]

in DA, PA, TA
out DC, PC, TC
var $DDSt, PDSt, TDSt$

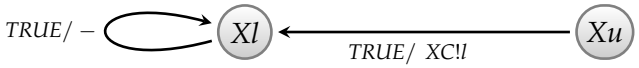
$F_{11,d} \otimes F_{11,p} \otimes F_{11,d}$

[$F_{11,X}$ with $X \in d, p, t$]

in

out *XC*

var *XDSt*



UC12

Short description: *Driver and passenger doors only react on the general closing trigger if the key is at the trunk.*

Context:

- $KPos = tr$

Preconditions:

- –

Actions:

1. *A person executes a trunk door action*

Postcondition:

- *Passenger and Driver doors remain in their state.*

[$F_{12,X}$ with $X \in \{P, D\}$]

in *TA*

out *XC*

var *XDSt*

UC13

Short description: *Switch to protection mode*

Requirement:

- *If the opening gesture is recognized maximal 5 times without a valid key at the back opening by gesture is deactivated.*
- *If the motor is started the protection mode is left and the system returns to normal operation.*

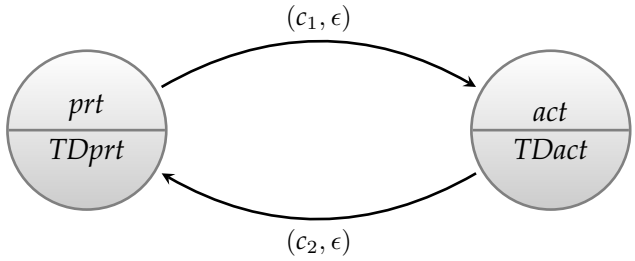
This requirement is different. It does not include nor refer to any behavior that is directly observable. It just regulates a certain mode change.

[M_{13}]

in $\{TA, KPos\} \cup I_{TDprt} \cup I_{TDact}$

out $O_{TDprt} \cup O_{TDact}$

var $\{count, PC_{13}, h_{13}\} \cup V_{TDprt} \cup V_{TDact}$



c_1						
pre		*in*			*post*	
count	PC_{13}	*TA*	*KPos*	*ER*	*count*	PC_{13}
≤ 4	*act*	*t*	*tr*	*	*0*	*act*
≤ 4	*	*	*	*on*	*0*	*act*
<4	*act*	*t*	$\neg$ *tr*	*	*count+1*	*act*
=4	*act*	*t*	$\neg$ *tr*	$\neg$ *on*	*5*	*prt*
c_2						
count	PC_{13}	*TA*	*KPos*	*ER*	*count*	PC_{13}
*	*prt*	*	*	$\neg$ *on*	*5*	*prt*
*	*prt*	*	*	*on*	*0*	*act*

B.2. Requirements CRCs

We omit the reference to the scenarios because we do not use them.

Name		Scope		Requirement
		IN	OUT	
VHigh		$V = hi$	$V = low$	
	1	$V = hi$	$V = low$	F_1
VLow		$V = low$	$V = hi$	
No Key		$KPos = away$	$KPos = \overline{away}$	
	3	$UBat = hi$	$UBat = hi$	F_3
	9			F_9
Key@FrtDoors		$KPos \in dd, pd$	$KPos \notin dd, pd$	
	5,D			$F_{5,D}$
	5,P			$F_{5,P}$
	9			F_9
	8,D(a-d)			$F_{8,D(a-d)}$
	10,D			$F_{10,D}$
	8,Pb			$F_{8,Pb}$
	10,P			$F_{10,P}$
	8,Tb			$F_{8,Tb}$
	10,T			$F_{10,T}$
Key@Trunk		$KPos = tr$	$KPos \neq tr$	
	12,D			$F_{12,D}$
	9,D			$F_{9,D}$
	12,P			$F_{12,P}$
	9,P			$F_{9,P}$
Bat Low		$UBat = low$	$UBat = hi$	
	4			F_4
	6			F_6
	7			F_7
	9			F_9

Bat High	$UBat = hi$	$UBat = low$	
Protect	c_2	c_1	
4			F_4
6			F_6
7			F_7
9			F_9
Act	c_1	c_2	
2			F_2
3			F_3
4			F_4
6			F_6
9			F_9

B.3. Service hierarchy

As Section 5.5 describes the car entry system can be embedded into a component architecture. Such a step includes the linking of the input "gesture" with the output of an upstreamed component's output that does the actual detection (and can be replaced if new technology becomes available). Figure B.1 illustrates the integration of two such components. Just the channel "gesture" is as required any other channels are only hinted.

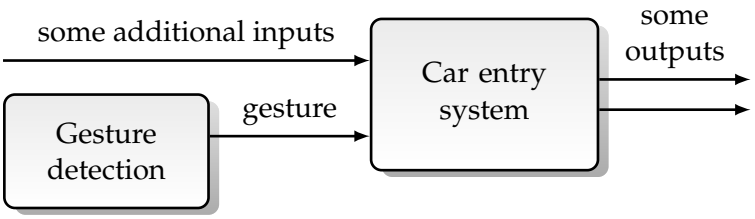
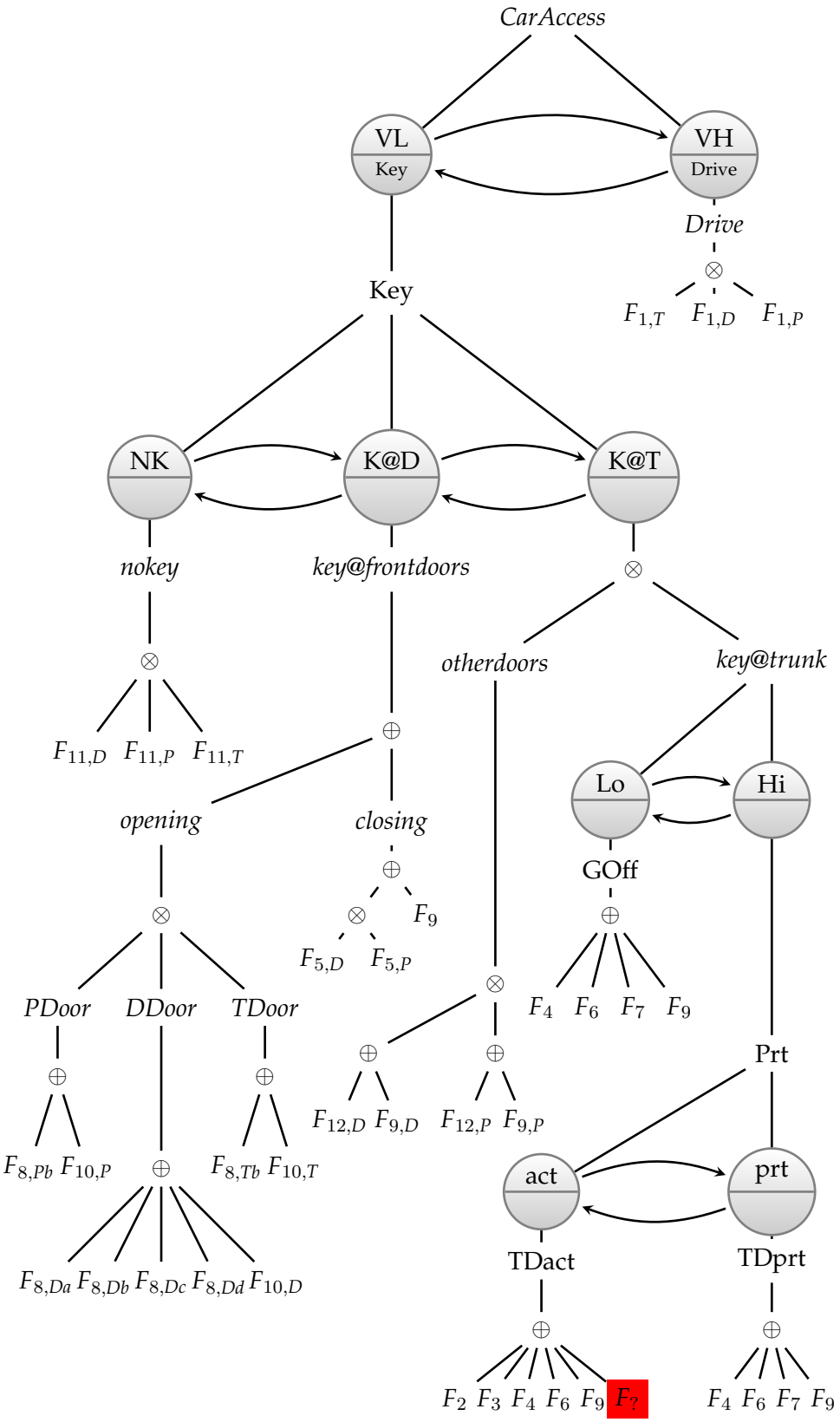


Figure B.1.: Integration of the car entry system with an upstreamed component for the actual gesture detection allowing to abstract from the gesture detection.

The decomposition of a system (or service) in a set of sub-services is a matter of engineering. There are different criteria that can be used, often depending on the application domain. Sometimes functional groups already exist.


CarAccess
VL
Key
VH
Drive
Drive
Key
$F_{1,T}$ $F_{1,D}$ $F_{1,P}$
NK
K@D
K@T
nokey
key@frontdoors
otherdoors
key@trunk
$F_{11,D}$ $F_{11,P}$ $F_{11,T}$
opening
closing
Lo
Hi
GOff
F_9
$F_{5,D}$ $F_{5,P}$
F_4 F_6 F_7 F_9
PDoor
DDoor
TDoor
Prt
$F_{12,D}$ $F_{9,D}$ $F_{12,P}$ $F_{9,P}$
$F_{8,Pb}$ $F_{10,P}$
$F_{8,Tb}$ $F_{10,T}$
act
prt
$F_{8,Da}$ $F_{8,Db}$ $F_{8,Dc}$ $F_{8,Dd}$ $F_{10,D}$
TDact
TDprt
F_2 F_3 F_4 F_6 F_9 $F_?$
F_4 F_6 F_7 F_9

In the hierarchy we organized the contexts hierarchically according to the contextual requirements chunks and some small optimizations that are reasonable like, e.g., manual unbundling.

The service hierarchy is yet incomplete. There are some obvious services missing: in some modes the service hierarchy does not describe the behavior of some doors effectively letting them behave arbitrarily. Additionally the service hierarchy does not include the services that model feature interactions as identified in Chapter 5.

Subsequently we present the set of defining equations. It should be easy to see the correspondence between the equations and the service hierarchy.

$$System \stackrel{def}{=} EngCrtl \otimes CarAccess \quad //MainSystemSpec \tag{B.1}$$

$$EngCrtl \stackrel{def}{=} \text{Not defined in case study} \quad //EngineControl \tag{B.2}$$

$$\begin{aligned} CarAccess \stackrel{def}{=} & (\{VL, VH\}, \\ & \{(VH, (V = lo, \epsilon), VL), \\ & \ (VL, (V = hi, \epsilon), VH)\}, \\ & \{(VL \mapsto Key), (VH \mapsto Drive)\}) \end{aligned} \quad //CarAccess \tag{B.3}$$

$$Drive \stackrel{def}{=} F_{1,t} \otimes F_{1,d} \otimes F_{1,p} \quad //CarDriving \tag{B.4}$$

$$\begin{aligned} Key \stackrel{def}{=} & (\{NK, K@D, K@T\}, \\ & \{(NK, (KPos \in \{dd, pd\}, \epsilon), K@D), \\ & \ (NK, (KPos = tr, \epsilon), K@T), \\ & \ (K@D, (KPos = away, \epsilon), NK), \\ & \ (K@D, (KPos = tr, \epsilon), K@T), \\ & \ (K@T, (KPos \in \{dd, pd\}, \epsilon), K@D), \\ & \ (K@T, (KPos = away, \epsilon), NK)\} \\ & \{(NK \mapsto nokey), \\ & \ (K@D \mapsto Key@frontdoors), \\ & \ (K@T \mapsto Key@trunk)\}) \end{aligned} \quad //FindingKeys \tag{B.5}$$

$$nokey \stackrel{def}{=} F_{11,D} \oplus F_{11,P} \oplus F_{11,T} \quad //NoKey \tag{B.6}$$

$$Key@frontdoors \stackrel{def}{=} opening \oplus closing \quad //Doorspeckeyatfront \tag{B.7}$$

$$closing \stackrel{def}{=} (F_{5,D} \otimes F_{5,P}) \oplus F_9 \quad //Closingdoors \tag{B.8}$$

$$opening \stackrel{def}{=} DDoor \otimes PDoor \otimes TDoor \quad //OpeningDoors \tag{B.9}$$

$$TDoor \stackrel{def}{=} F_{8,Tb} \oplus F_{10,T} \quad //PassengerDoor \tag{B.10}$$

$$DDoor \stackrel{def}{=} F_{8,Da} \oplus F_{8,Db} \oplus F_{8,Dc} \oplus F_{8,Dd} \oplus F_{10,T} \quad //DriverDoor \tag{B.11}$$

$$PDoor \stackrel{def}{=} F_{8,Pb} \oplus F_{10} \quad //PassengerDoor \tag{B.12}$$

$$aux_1 \stackrel{def}{=} otherdoors \otimes Keyattrunk \quad //Auxiliaryhierarchylevel \tag{B.13}$$

$$otherdoors \stackrel{def}{=} (F_{12,D} \oplus F_{9,D}) \otimes (F_{12,P} \oplus F_{9,P}) \quad //behaviorofTDandPD \tag{B.14}$$

$$\begin{aligned} Key@trunk \stackrel{def}{=} \; & (\{Hi, Lo\}, \\ & \{(Hi, (UBat = lo, \epsilon), Lo), \\ & \; (Lo, (UBat = hi, \epsilon), HI)\}, \\ & \{(Hi \mapsto Prt), (Lo \mapsto GOff)\}) \end{aligned} \quad //TrunkSpec \tag{B.15}$$

$$GOff \stackrel{def}{=} F_4 \oplus F_6 \oplus F_7 \oplus F_9 \quad //GesturesOff \tag{B.16}$$

$$\begin{aligned} Prt \stackrel{def}{=} \; & (\{act, prt\}, \\ & \{(act, (c_1, \rho), prt), (prt, (c_2, \rho), act)\}, \\ & \{(prt \mapsto TDprt), (act \mapsto TDact)\}) \end{aligned} \quad //Protection \tag{B.17}$$

$$TDact \stackrel{def}{=} F_2 \oplus F_3 \oplus F_4 \oplus F_6 \oplus F_9 \oplus DDTRFI \quad //Gesturesensitivity \tag{B.18}$$

$$TDprt \stackrel{def}{=} F_4 \oplus F_6 \oplus F_7 \oplus F_9 \quad //Nogesturesensitivity \tag{B.19}$$

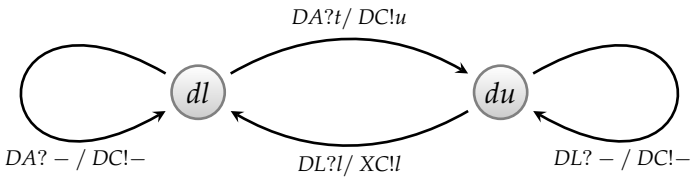
(B.20)

B.4. Illustration of intermediate steps and restructuring

The service tree already uniquely defines the behavior of the service. Nevertheless, we present some selected behaviors resulting from combinations of sub-services after restructuring. We add behaviors according to identifying and removing deficiencies

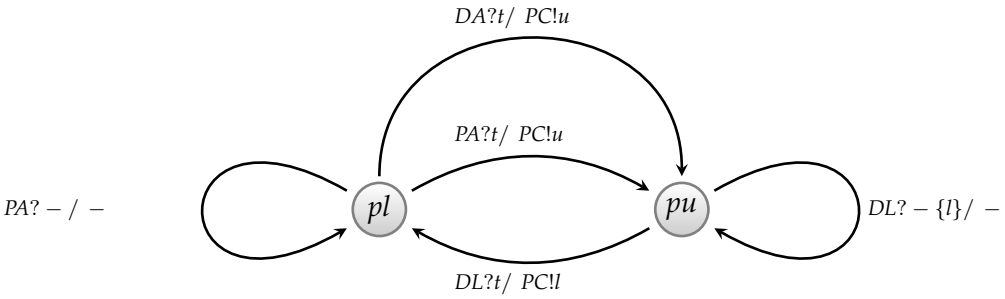
B.5. Driver door

B.5.1. Low speed, key available



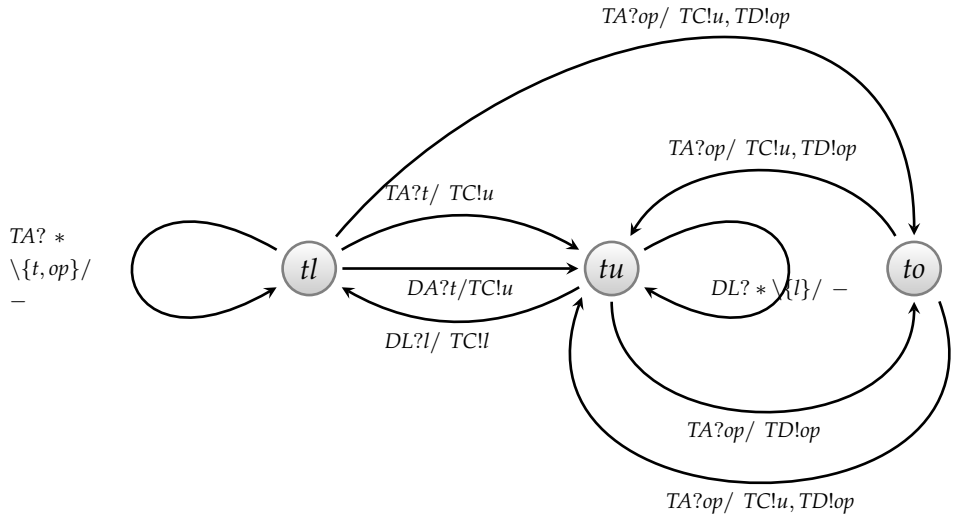
B.6. Passenger door

B.6.1. Low speed, key available



B.7. Trunk Door

B.7.1. Low speed, key available, battery high



Bibliography

Abadi, M. and Lamport, L. (1995). Conjoining specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(3):507–535.

Aksit, M. and Choukair, Z. (2003). Dynamic, adaptive and reconfigurable systems overview and prospective vision. *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on*, pages 84–89.

Alpaydin, E. (2004). *Introduction To Machine Learning*. MIT Press.

Alur, R. and Grosu, R. (2000). Modular refinement of hierarchic reactive machines. In *In Proceedings of the 27th Annual ACM Symposium on Principles of Programming Languages*, pages 390–402. ACM Press.

Astesiano, E., Broy, M., and Reggio, G. (1999). Algebraic specification of concurrent systems.

Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking*. The MIT Press.

Baldauf, M., Dustdar, S., and Rosenberg, F. (2007). A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263.

Balser, M. (2005). *Verifying Concurrent Systems with Symbolic Execution : Temporal Reasoning is Symbolic Execution with a Little Induction*. PhD thesis, Universität Augsburg, Universitätsstr. 22, 86159 Augsburg.

Beckert, B., Hahnle, R., and Escalada-Imaz, G. (1998). Simplification of many-valued logic formulas using anti-links. *Journal of Logic and Computation*, 8(4):569.

Berardi, D., Calvanese, D., De Giacomo, G., Hull, R., and Mecella, M. (2005). Automatic Composition of Transition-based Semantic Web Services with Messaging. In *Proc. of the 31st Int. Conf. on Very Large Data Bases (VLDB 2005)*, pages 613–624.

Bergstra, J. and Klop, J. (1985). Algebra of communicating processes with abstraction. *THEORET. COMP. SCI.*, 37(1):77–121.

Bergstra, J., Klop, J., and Olderog, E. (1988). Readies and Failures in the Algebra of Communicating Processes. *SIAM Journal on Computing*, 17:1134.

Berry, G. and Gonthier, G. (1992). The ESTEREL synchronous programming language: design, semantics, implementation. *Science of computer programming*, 19(2):87–152.

Berry, G. and Sethi, R. (1986). From regular expressions to deterministic automata. *Theoretical computer science*, 48:117–126.

Biegel, G. and Cahill, V. (2004). A Framework for Developing Mobile, Context-aware Applications. In *PERCOM '04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*, page 361, Washington, DC, USA. IEEE Computer Society.

Bittner, K. and Spence, I. (2002). *Use Case Modeling*. Addison-Wesley.

Boehm, B. (1981). Software engineering economics. *Englewood Cliffs*.

Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., and Orchard, D. (2004). Web Services Architecture. *W3C Working Group Note*, 11:2005–1.

Botaschanjan, J., Harhurin, A., and Kof, L. (2008). Service-Based Specification of Reactive Systems. techreport, Technische Universität München.

Bradfield, J. and Stirling, C. (2001). Modal logics and mu-calculi. In Bergstra, J., Ponse, A., and Smolka, S., editors, *Handbook of Process Algebra*, pages 293–332. Elsevier, North-Holland.

Brown, P. (1996). The stick-e document: a framework for creating context-aware applications. *Electronic Publishing@article*, 96:259–272.

Broy, M. (1997). The Specification of System Components by State Transition Diagrams. Technical Report TUM-I9729, Technische Univerität München.

Broy, M. (2005). *Engineering Theories of Software Intensive Systems*, chapter Service-oriented Systems Engineering: Specification and Design of Services and Layered Architectures - The JANUS Approach, pages 47 – 81. Springer Verlag.

Broy, M. (2007). Model-driven architecture-centric engineering of (embedded) software intensive systems: modeling theories and architectural milestones. *Innovations in Systems and Software Engineering*, 3, Number 1:75–102.

Broy, M. (2010). Multifunctional software systems: Structured modeling and specification of functional requirements. *Science of Computer Programming*, In Press, Corrected Proof:–.

Broy, M., Feilkas, M., Grünbauer, J., Gruler, A., Harhurin, A., Hartmann, J., Penzenstadler, B., Schätz, B., and Wild, D. (2008). Umfassendes Architekturmodell für das Engineering eingebetteter Software-intensiver Systeme. Technical Report TUM-I0816, Technische Universität München.

Broy, M., Krüger, I. H., and Meisinger, M. (2007). A formal model of services. *ACM Trans. Softw. Eng. Methodol.*, 16(1):5.

Broy, M., Leuxner, C., Sitou, W., Spanfelner, B., and Winter, S. (2009). Formalizing the notion of adaptive system behavior. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1029–1033, New York, NY, USA. ACM.

Broy, M. and Stølen, K. (2001). *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer.

Bruegge, B. and Dutoit, A. H. (2003). *Object-Oriented Software Engineering Using UML, Patterns, and Java*. Prentice Hall, Englewood Cliffs, NJ, second edition.

Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J. (1987). LUSTRE: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188. ACM.

Chen, G. and Kotz, D. (2000). A Survey of Context-Aware Mobile Computing Research. Technical Report TR2000-381, Dartmouth Computer Science.

Chen, H. (2004). *An Intelligent Broker Architecture for Pervasive Context-Aware Systems*. PhD thesis, University of Maryland, Baltimore County.

Chen, H., Finin, T., , and Joshi, A. (2003). Using OWL in a Pervasive Computing Broker. In *Proceedings of Workshop on Ontologies in Open Agent Systems (AAMAS 2003)*.

Chen, P. P.-S. (1976). The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36.

Chinnici, R., Moreau, J., Ryman, A., and Weerawarana, S. (2004). Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. *W3C Working Draft*, 26.

Church, A. (1996). *Introduction to mathematical logic*. Princeton Univ Pr.

Clarke, E., Grumberg, O., and Peled, D. (1999). *Model checking*. Springer.

Coalition, T. (2004). Owl-s: Semantic markup for web services. *W3C Member Submission*, 22.

Cockburn, A. (2000). *Writing Effective Use Cases*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Connolly, D., van Harmelen, F., Horrocks, I., McGuinness, D., Patel-Schneider, P., and Stein, L. (2001). Daml+ oil (march 2001) reference description. *W3c note*, 18.

Coutaz, J., Crowley, J. L., Dobson, S., and Garlan, D. (2005). Context is key. *Commun. ACM*, 48(3):49–53.

Crowley, J., Coutaz, J., Rey, G., and Reignier, P. (2002). Perceptual components for context aware computing. *Lecture notes in computer science*, pages 117–134.

Da, T. and Zhang, Q. (2004). A middleware for building context-aware mobile services. In *2004 IEEE 59th Vehicular Technology Conference, 2004. VTC 2004-Spring*, volume 5.

Dahl, O., Dijkstra, E., and Hoare, C. (1972). *Structured programming*. ACM Classic Books Series.

Darimont, R., Delor, E., Massonet, P., and van Lamsweerde, A. (1997). GRAIL/KAOS: an environment for goal-driven requirements engineering. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 612–613, New York, NY, USA. ACM.

de Boer, F., Klop, J., and Palamidessi, C. (1992). Asynchronous communication in process algebra. In *Logic in Computer Science, 1992. LICS '92., Proceedings of the Seventh Annual IEEE Symposium on*, pages 137–147.

Dennett, D. C. (1984). Cognitive Wheels: The Frame Problem of AI. *Minds, Machines, and Evolution*, pages 128–151. Cambridge University Press.

Deubler, M. (2008). *Dienst-orientierte Softwaresysteme: Anforderungen und Entwurf*. PhD thesis, PhD thesis, Technische Universitat Munchen.

Dey, A. (1998). Context-Aware Computing: The CyberDesk Project. In *Proceedings of AAAI '98 Spring Symposium on Intelligent Environments*, pages 51–54.

Dey, A., Salber, D., and Abowd, G. (2001). A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16:97–166.

Dey, A. K. (2000). *Providing architectural support for building context-aware applications*. PhD thesis, College of Computing, Georgia Institute of Technology. Director-Gregory D. Abowd.

Dijkstra, E. (1968). A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8(3):174–186.

Fahrmair, M. (2005). *Kalibrierbare Kontextadaption für Ubiquitous Computing*. PhD thesis, Technische Universität München.

Fahrmair, M., Sitou, W., and Spanfelner, B. (2006a). An Engineering Approach to Adaptation and Calibration. In Roth-Berghofer, T. R., Schulz, S., and Leake, D. B., editors, *Modeling and Retrieval of Context*, volume 3946 of *Springer LNCS*, pages 134 – 147. Springer Berlin / Heidelberg LNCS 3946.

Fahrmair, M., Sitou, W., and Spanfelner, B. (2006b). Unwanted Behavior and its Impact on Adaptive Systems in Ubiquitous Computing. In *14th Workshop on Adaptivity and User Modeling in Interactive Systems – LWA/ABIS 2006*.

Fairley, R. (1985). *Software engineering concepts*. McGraw-Hill, Inc. New York, NY, USA.

Geihs, K. (2008). Selbst-adaptive Software. *Informatik-Spektrum*, 31(2):133–145.

Grosu, R., Klein, C., Rumpe, B., and Broy, M. (1996). State Transition Diagrams. techreport, Technische Universität München.

Gruler, A. and Meisinger, M. (2009). Hierarchical Decomposition of Multi-Functional Systems. Technical Report TUM-I0901, Technische Universität München.

Gu, T., Pung, H., and Zhang, D. (2005). A service-oriented middleware for building context-aware services. *Journal of Network and Computer Applications*, 28(1):1–18.

Gudemann, M., Nafz, F., Ortmeier, F., Seebach, H., and Reif, W. (2008). A specification and construction paradigm for organic computing systems. In *Self-Adaptive and Self-Organizing Systems, 2008. SASO'08. Second IEEE International Conference on*, pages 233–242. IEEE.

Gunter, C., Gunter, E., Jackson, M., and Zave, P. (2002). A reference model for requirements and specifications. *Software, IEEE*, 17(3):37–43.

Halpin, T. (1996). Business Rules and Object Role Modeling. *Database Programming & Design*, 9(10):66 – 72.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274.

Harhurin, A. (2010). *Von separaten Interaktionsmustern zu konsistenten Spezifikationen reaktiver Systeme*. Dissertation, Technische Universität München, München.

Harrison, J. (2009). *Handbook of practical logic and automated reasoning*. Cambridge University Press New York, NY, USA.

Hehner, E. C. R. (1984a). Predicative programming Part I. *Commun. ACM*, 27(2):134–143.

Hehner, E. C. R. (1984b). Predicative programming Part II. *Commun. ACM*, 27(2):144–151.

Heitmeyer, C. L., Jeffords, R. D., and Labaw, B. G. (1996). Automated consistency checking of requirements specifications. *ACM Trans. Softw. Eng. Methodol.*, 5(3):231–261.

Henricksen, K., Indulska, J., and McFadden, T. (2005). Modelling Context Information with ORM. In *OTM Federated Conferences Workshop on Object-Role Modeling (ORM)*, volume 3762 of *LNCS*, pages 626–635. Springer.

Henriksen, J., Jensen, J., Jørgensen, M. andKlarlund, N., Paige, B., Rauhe, T., and Sandholm, A. (1995). Mona: Monadic Second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems,First International Workshop, TACAS '95, LNCS 1019*.

Henzinger, T. A. (2000). Masaccio: A Formal Model for Embedded Components. In *TCS '00: Proceedings of the International Conference IFIP on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics*, pages 549–563, London, UK. Springer-Verlag.

Heun, V. (2003). *Grundlegende Algorithmen: Einf "uhrung in den Entwurf und die Analyse effizienter Algorithmen*. Vieweg+ Teubner Verlag.

Hoare, C. (1985). *Communicating sequential processes*. Prentice/Hall International.

Hoare, C. A. R. (1978). Communicating sequential processes. *Commun. ACM*, 21(8):666–677.

Hoare, C. A. R. (1997). Unified theories of programming. In *Mathematical methods in program development*, volume 158, pages 313–367. NATO advanced study institute on mathematical methods in program development, Marktoberdorf.

Huber, F., Schätz, B., and Einert, G. (1997). Consistent Graphical Specification of Distributed Systems. In Fitzgerald, J., Jones, C. B., and Lucas, P., editors, *FME '97: 4th International Symposium of Formal Methods Europe, Lecture Notes in Computer Science 1313*, pages 122 – 141. Springer.

IEEE (1998). IEEE Recommended Practice for Software Requirements Specifications. Technical report, Institute of Electrical and Electronics Engineers, Inc.

Jackson, M. and Zave, P. (1998). Distributed feature composition: a virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, 24:831–847.

Janin, D. and Walukiewicz, I. (1996). On the expressive completeness of the propositional mu-calculus with respect to monadic second order logic. *CONCUR'96: Concurrency Theory*, pages 263–277.

Kahn, G. (1988). Natural Semantics. In *Proceedings of the first Franco-Japanese Symposium on Programming of future generation computers*, pages 237–257, Amsterdam, The Netherlands, The Netherlands. Elsevier Science Publishers B. V.

Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, A. (1990). Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie Mellon University.

Kang, K. C., Kim, S., Lee, J., Kim, K., Shin, E., and Huh, M. (1998). FORM: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.*, 5:143–168.

Kephart, J. and Chess, M. (2003). The vision of autonomic computing. *Computer*, 1(36):41–50.

Klarlund, N. and Møller, A. (2001). *Mona version 1.4: User manual*. Citeseer.

Knuth, D. E. (1964). Backus normal form vs. backus naur form. *Communications of the ACM*, 7(12):735–736.

Kolos-Mazuryk, L., van Eck, P. A. T., and Wieringa, R. J. (2006). A Survey of Requirements Engineering Methods for Pervasive Services. Deliverable TI/RS/2006/018, Freeband A-MUSE, Enschede.

Krüger, I. (2000). *Distributed System Design with Message Sequence Charts*. PhD thesis, Institut für Informatik, Technische Universität München.

Krüger, I., Grosu, R., Scholz, P., and Broy, M. (1999). From MSCs to statecharts. In *DIPES '98: Proceedings of the IFIP WG10.3/WG10.5 international workshop on Distributed and parallel embedded systems*, pages 61–71, Norwell, MA, USA. Kluwer Academic Publishers.

Kugele, S., Tautschnig, M., Bauer, A., Schallhart, C., Merenda, S., Haberl, W., Kühnel, C., Müller, F., Wang, Z., Wild, D., Rittmann, S., and Wechs, M. (2007). COLA – The Component Language. techreport, TU-München.

Leroy, X. (2010). *Mechanized semantics*, volume 25 of *NATO Science for Peace and Security Series D: Information and Communication Security*, pages 195–224. IOS Press.

Leuxner, C., Sitou, W., and Spanfelner, B. (2010). A formal model for work flows. In *Proceedings of the 8th International Conference on Software Engineering and Formal Methods*.

Leuxner, C., Sitou, W., Spanfelner, B., Schneider, A., Feussner, H., and Broy, M. (2009a). Towards Context-Aware Surgery Assistance for Laparoscopic Cholecystectomy. In *Proceedings of the 5th Russian-Bavarian Conference on Biomedical Engineering*, pages 121–123, Munich, Germany. Mitigroup.

Leuxner, C., Sitou, W., Spanfelner, B., Thurner, V., and Schneider, A. (2009b). Modeling Work Flows For Building Context-Aware Applications. techreport, Technische Universität München, Institut für Informatik.

Lynch, N. A. and Tuttle, M. R. (1989). An introduction to input/output automata. *CWI Quarterly*, 2:219–246.

Manna, Z. and Pnueli, A. (1995). *Temporal verification of reactive systems: safety*. Springer Verlag.

Maraninchi, F. and Rémond, Y. (1998). Mode-automata: About modes and states for reactive systems. *Programming Languages and Systems*, pages 105–115.

McCarthy, J. and Buvac, S. (1998). Formalizing context (expanded notes). *Computing natural language*, 81:13–50.

McCluskey, E. J. (1956). Minimization of Boolean functions. *The Bell System Technical Journal*, 35(5):1417–1444.

Mealy, G. (1955). A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079.

Meisinger, M. and Rittmann, S. (2008). A comparison of service-oriented development approaches. techreport, Technische Universität München.

Meyer, B. (1988). *Object-oriented software construction*, volume 59. Citeseer.

Milner, R. (1982). *A calculus of communicating systems*. Springer-Verlag New York, Inc. Secaucus, NJ, USA.

Milner, R. (1999). *Communicating and mobile systems: the π-calculus*. Cambridge Univ Pr.

Minsky, M. and Papert, S. (1988). *Perceptrons: An Introduction to Computational Geometry*. The MIT Press.

Mohyeldin, E., Fahrmair, M., Sitou, W., and Spanfelner, B. (2005). A Generic Framework for Context Aware and Adaptation Behaviour of Reconfigurable Systems. In *The 16th Annual IEEE International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC05), 11-14 September 2005, Berlin, Germany*.

Moore, E. (1956). Gedanken-experiments on sequential machines. *Automata studies*, 34:129–153.

Moszkowski, B. (1985). A Temporal Logic for Multilevel Reasoning about Hardware. *Computer*, 18(2):10–19.

Müller-Schloer, C. (2004). Organic computing: on the feasibility of controlled emergence. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 2–5. ACM.

Nafz, F., Ortmeier, F., Seebach, H., Stegh
"ofer, J., and Reif, W. (2009). A universal self-organization mechanism for role-based organic computing systems. *Autonomic and Trusted Computing*, pages 17–31.

Nafz, F., Seebach, H., Stegh
"ofer, J., B
"aumler, S., and Reif, W. (2010). A Formal Framework for Compositional Verification of Organic Computing Systems. *Autonomic and Trusted Computing*, pages 17–31.

Nazareth, D., Regensburger, F., and Scholz, P. (1996a). Mini-statecharts: A compositional way to model parallel systems. In *9th International Conference on Parallel and Distributed Computing Systems (PDCS'96)*. Citeseer.

Nazareth, D., Regensburger, F., and Scholz, P. (1996b). Mini-Statecharts A Lean Version of Statecharts. Technical Report TUM-I9610, Technische Univerität München.

Nipkow, T., Paulson, L., and Wenzel, M. (2002). *Isabelle/HOL: a proof assistant for higher-order logic*. Springer Verlag.

Olderog, E.-R. (1985). Semantics of Concurrent Processes: The Search for Structure and Abstraction - Part I. *EACTS Bulletins*, 28:73–97.

Olderog, E.-R. (1986). Semantics of Concurrent Processes: The Search for Structure and Abstraction - Part II. *EACTS Bulletins*, ?:96–117.

Oreizy, P., Gorlick, M. M., Taylor, R. N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D. S., and Wolf, A. L. (1999). An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62.

Palmer, E. (1995). Oops, it didn't arm'- A Case Study of two Automation Surprises. In *8th International Symposium on Aviation Psychology, Columbus, OH, United States*, pages 227–232.

Penzenstadler, B. (2011). *DeSyRe - Decomposition of Systems and their Requirements*. PhD thesis, TU-München.

Pulvermueller, E., Speck, A., Coplien, J. O., D'Hondt, M., and DeMeuter, W. (2002). Feature Interaction in Composed Systems. *LNCS*, 2323:86–97.

Rausch, A. and Niebuhr, D. (2005). Erfolgreiche IT-Projekte mit dem V-Modell XT. *OBJEKTspektrum*, 3(2005):42–49.

Reiter, R. (1987). On closed world data bases. In *Readings in nonmonotonic reasoning*, page 310. Morgan Kaufmann Publishers Inc.

Richardson, D. (1968). Some undecidable problems involving elementary functions of a real variable. *Journal of Symbolic Logic*, 33(4):514–520.

Rittmann, S. (2008). *A methodology for modeling usage behavior of multi-functional systems*. PhD thesis, Technische universität München.

Sarter, N., Woods, D., and Billings, C. (1997). Automation surprises. *Handbook of Human Factors and Ergonomics*, 2:1926–1943.

Schaefer, I. (2008). *Integrating Formal Verification into the Model-Based Development of Adaptive Embedded Systems*. Verl. Dr. Hut.

Schaetz, B. (2002). Towards Service-Based Systems Engineering: Formalizing and mu-Checking Service Descriptions. techreport, Technische Universität München.

Schätz, B. (2006). Building Components from Functions. *Electr. Notes Theor. Comput. Sci.*, 160:321–334.

Schätz, B. (2007). Modular Functional Descriptions. In *Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2007)*.

Schätz, B. (2009). *Model-Based Development of Software Systems: From Models to Tools*. PhD thesis, Technische Universität München.

Schätz, B., Romberg, J., Slotosch, O., Strecker, M., Wišpeintner, A., Hain, T., Prenninger, W., Rappl, M., and Spies, K. (2003). Modeling Embedded Software: State of the Art and Beyond. In *Proceedings of ICCSEA 16th International Conference on Software and Systems Engineering and their Applications, 2003 Conference*.

Schätz, B. and Salzmann, C. (2003). Service-Based Systems Engineering: Consistent Combination of Services. In *Proceedings of ICFEM 2003*.

Schilit, B., Adams, N., and Want, R. (1994). Context-Aware Computing Applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, US.

Schilit, W. N. (1995). *A System Architecture for Context-Aware Mobile Computing*. PhD thesis, Columbia University.

Schmidt, A., Beigl, M., and Hans-W, H. (1999). There is more to context than location. *Computers and Graphics*, 23(6):893–901.

Schoning, U. (2000). *Logik fur Informatiker*, volume 56. Spektrum Akademischer Verlag.

Sirin, E., Hendler, J., , and Parsia, B. (2003). Semi-automatic composition of web services using semantic descriptions. In *Proc. of Web Services: Modeling, Architecture and Infrastructure workshop in conjunction with ICEIS2003*.

Sitou, W. (2009). *Requirements Engineering kontextsensitiver Anwendungen*. Dissertation, Technische Universität München.

Sitou, W. and Spanfelner, B. (2007). Towards Requirements Engineering for Context Adaptive Systems. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, volume 2, pages 593–600, Beijing, China. IEEE Computer Society.

Spichkova, M. (2007). *Specification and Seamless Verification of Embedded Real-Time Systems: FOCUS on Isabelle*. PhD thesis, Technische Universität München.

Srivastava, B. and Koehler, J. (2003). Web service composition — current solutions and open problems. In *ICAPS 2003*.

Stølen, K. (1996). Refinement Principles Supporting the Transition from Asychronous to Synchronous Communications. *Science of Computer Programming*.

Sutcliffe, A., Fickas, S., and Sohlberg, M. (2005). Personal and contextual requirements engineering. In *Proceedings. 13th IEEE International Conference on Requirements Engineering, 2005.*, pages 19 – 28.

Sutcliffe, A., Fickas, S., and Sohlberg, M. M. (2006). PC-RE: a method for personal and contextual requirements engineering with some experience. *Requirements Engineering*, 11(No. 4):157–173.

Thomas, W. (1990). Automata on infinite objects. *Handbook of theoretical computer science*, pages 133–191.

Trapp, M. (2005). *Modeling the Adaptation Behavior of Adaptive Embedded Systems*. PhD thesis, PhD thesis, University of Kaiserslautern.

van Lamsweerde, A. et al. (2001). Goal-Oriented Requirements Engineering: A Guided Tour. *Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, page 249.

Vardi, M. and Wolper, P. (1986). *An automata-theoretic approach to automatic program verification*. International Business Machines Inc., Thomas J. Watson Research Center.

von der Beeck, M. (1994). A Comparison of Statecharts Variants. *Proceedings of the Third International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 128–148.

Want, R., Hopper, A., and Gibbons, V. (1992). The Active Badge Location System. *ACM Transactions on Information Systems*, 10(1):91–102.

Weiser, M. (September 1991). The computer for the 21st century. *Scientific American*, 265(3):94–104.

Winograd, T. (2001). Architectures for context. *Human-Computer Interaction*, 16(2):401–419.

Winter, S. (2009). *Modellbasierte Analyse von Nutzerschnittstellen*. Dissertation, Technische Universität München, München.

Wu, D., Sirin, E., Parsia, B., Hendler, J., and Nau, D. (2003). Automatic web services composition using SHOP2. In *Proceedings of Planning for Web Services Workshop in ICAPS 2003*, Trento, Italy.

Yu, E. S. K. (1997). Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering. In *3rd IEEE Intl. Symposium on in Requirements Engineering*, pages 226–235.

Zave, P. (1993). Feature Interactions and Formal Specifications in Telecommunications. *Computer*, 26(8):20–29.

Zave, P. (2001). Feature-oriented description, formal methods, and DFC. In Gilmore, S. and Ryan, M., editors, *Language Constructs for Describing Features*, Proceedings of the FIREworks workshop, pages 11–26. Springer.

Zave, P. and Jackson, M. (2000). New Feature Interactions in Mobile and Multimedia Telecommunications Services. In Calder, M. and Magill, E. H., editors, *Feature Interactions in Telecommunications and Software Systems VI*, pages 51–66. IOS Press.