SN Computer Science (2023) 4:151
https://doi.org/10.1007/s42979-022-01532-z

ORIGINAL RESEARCH

The Universal Safety Format in Action: Tool Integration and Practical Application

Frederik Haxel[1] · Alexander Viehl[1] · Michael Benkel[2] · Bjoern Beyreuther[2] · Klaus Birken[3] · Rolf Schmedes[4] · Kim Grüttner[4] · Daniel Mueller-Gritschneder[5]

Received: 22 May 2022 / Accepted: 25 November 2022 / Published online: 9 January 2023

Abstract

Designing software that meets the stringent requirements of functional safety standards imposes a significant development effort compared to conventional software. A key aspect is the integration of safety mechanisms into the functional design to ensure a safe state during operation even in the event of hardware errors. These safety mechanisms can be applied at different levels of abstraction during the development process and are usually implemented and integrated manually into the design. This does not only cause significant effort but does also reduce the overall maintainability of the software. To mitigate this, we present the Universal Safety Format (USF), which enables the generation of safety mechanisms based on the separation of concerns principle in a model-driven approach. Safety mechanisms are described as generic patterns using a transformation language independent from the functional design or any particular programming language. The USF was designed to be easily integrated into existing tools and workflows that can support different programming languages. Tools supporting the USF can utilize the patterns in a functional design to generate and integrate specific safety mechanisms for different languages using the transformation rules contained within the patterns. This enables not only the reuse of safety patterns in different designs, but also across different programming languages. The approach is demonstrated with an automotive use-case as well as different tools supporting the USF.

Keywords Functional safety · Software safety mechanism · Model transformation · Code generation · Domain-specific language

Introduction

Safety-critical systems should never cause harm to people or property even in the presence of random hardware faults. Functional safety standards guide the development of such systems. Several standards exist for programmable

This article is part of the topical collection "Advances on Model-Driven Engineering and Software Development" guest edited by Luís Ferreira Pires and Slimane Hammoudi.

✉ Frederik Haxel
haxel@fzi.de

Alexander Viehl
viehl@fzi.de

Michael Benkel
benkel@scopeset.de

Bjoern Beyreuther
beyreuther@scopeset.de

Klaus Birken
klaus.birken@itemis.de

Rolf Schmedes
rolf.schmedes@dlr.de

Kim Grüttner
kim.gruettner@dlr.de

Daniel Mueller-Gritschneder
daniel.mueller@tum.de

[1] FZI Research Center for Information Technology, Karlsruhe, Germany

[2] ScopeSET GmbH, Fischbachau, Germany

[3] itemis AG, Stuttgart, Germany

[4] German Aerospace Center (DLR), Oldenburg, Germany

[5] Technical University of Munich, Munich, Germany

electronic systems, such as the general IEC 61508 [1] or domain-specific standards such as ISO 26262 [2] for the automotive domain. These standards specify additional development steps such as the implementation and integration of safety mechanisms. These mechanisms are technical solutions to detect faults or control failures in order to achieve or maintain a safe state, and can be implemented in software, hardware, or a combination of both. While historically many safety mechanisms have been implemented through additional hardware components, software safety mechanisms are becoming increasingly important due to the growth of software-intensive systems and the desire to use more commercial off-the-shelf hardware. Software safety mechanisms can be integrated into the functional design at different levels of abstractions, for example, in a model of the design [3, 4], in the source code [5], or at binary level [6–9]. The optimal abstraction level and modeling/programming language (hereafter referred to as *domain*) for the integration of each software safety mechanism depends on many aspects and is very application-specific. Therefore, it is not uncommon to use a combination of abstraction levels to integrate safety mechanisms. The development of safety mechanisms is only one part of the overall safety engineering process, but while there exist many methodologies and tools to support engineers in different parts of this process, the development of application-specific software safety mechanisms remains a predominantly manual process, which is prone to errors and time-consuming. However, these safety mechanisms can be often divided into different types of mechanisms that share a common structure regardless of the application and even across different domains [10]. Additionally, the integration of safety mechanisms into the functional design bloats the overall software, which not only increases the maintenance effort, but also tends to make it significantly more difficult to understand the functional software.

To automate this labor-intensive realization of software safety mechanisms for different domains, we introduced the Universal Safety Format (USF) methodology in [11]. In this methodology, safety mechanisms are generalized and described in the USF via patterns in a dedicated domain-independent transformation language. The patterns can then be used in a design by a domain-specific tool that supports the USF to implement the domain-agnostic transformation in a domain-specific context. This allows the user not only to generate the safety mechanisms, but also to keep the functional software separate from the safety mechanisms, as they can be integrated at any time. While [11] focused on the concept of USF, this paper explores the technical aspects of the approach in more detail. The main contributions are:

- A detailed description of the USF metamodel that forms the basis for the USF transformation language.
- A comprehensive explanation of the USF transformation language (UTL) used to integrate safety mechanism patterns into a model.
- An extended evaluation showing how the USF is integrated with various tools to apply safety mechanisms at different levels of abstraction.

The remainder of this work is structured as follows: Section "Safety Engineering in a Nutshell" provides a brief summary of the safety engineering process and introduces a running example. Section "USF in the Development Flow" outlines how the USF process can be incorporated into an established development workflow. A description of the USF metamodel is given in Section "USF Metamodel", whereas Section "USF Transformation Language" addresses the UTL. Section "Evaluation" presents several tools that support the USF as well as the application of safety mechanisms on the presented running example at a model and source code level. The results are discussed in Section "Discussion" and a comparison of our approach with related research as well as with existing modeling and transformation languages is given in Section "Related Work". Section "Conclusions" provides a summary of our findings and outlines future work.

Safety Engineering in a Nutshell

To minimize the risk of system failure, safety standards such as the aforementioned IEC 61508 [1] or one of its industry-specific adaptations such as the ISO 26262 [2] were elaborated. Those standards provide information on how to design, deploy and maintain a system for safety-related applications. For a better understanding of the topic of safety engineering, the following will take a closer look at the IEC 61508 standard, which assumes that every safety-related system must function or fail predictably and safely under all possible conditions.

IEC 61508 presents a comprehensive and holistic development process called the safety life cycle for the development of safety-related systems. The standard is structured in 16 phases, starting with analysis, continuing with the principles for realization and ending with phases on the operation of a system. The overarching goal of these phases is the correct execution of the safety-related functions. A fundamental part of this life cycle is the probabilistic failure approach, which classifies the safety impact that a failure of a component would have. It is part of the hazard and risk analysis, which consists of three phases: hazard identification, analysis assessment and risk assessment. For the risk assessment, risk is considered as a function of the probability of a hazardous event and the severity of its consequences. Either qualitative or quantitative analytical methods can be used to quantify the risk. This assessment provides information on which risks need to be reduced and thus enables an appropriate design of the protection system. Under-specification or over-specification thus become less likely.

To comply with the standard, the safety requirements must be provided with a target safety integrity level (SIL). The term safety integrity is defined by the standard as the probability that the safety-related system will satisfactorily perform the required safety functions under all stated conditions. There are four discrete safety integrity levels that define the safety integrity requirements of a function. The general reasoning behind SILs is then as follows: For a greater necessary risk reduction, the safety-related system needs to be more reliable, so the targeted SIL has to be higher.

IEC 61508 and its other related safety standards provide guidance on what safety mechanisms should be used to achieve a target SIL. Thus, there are a number of safety mechanisms, which are often used repeatedly in the development of safety-related systems. Some examples are: Error detection logic and codes, plausibility checks, range checks of input and output data, stack overflow/underflow detection, timing supervision with watchdogs, control flow monitoring and external monitoring facilities, static recovery mechanisms, hardware self-tests and majority voters.

To improve both the applicability and reusability of these safety mechanisms, the paper proposes the USF as a domain-agnostic specification format, which allows for the automatic generation and integration of safety mechanisms. The resulting capability to overcome these technical implementation hurdles more easily is the main advantage of USF. In addition to this, USF allows users of the IEC 61508 development process to map safety requirements to safety mechanism instances. This can be helpful in order to indicate whether a desired safety integrity level will be met or not.

Running Example

A running example of a simplified adaptive cruise control (ACC) system was chosen to demonstrate the potential of USF. Figure 1 shows the structural and functional composition of the example. The objective of an ACC is to control the speed of a vehicle so that it maintains a constant distance to a vehicle ahead. To achieve this, the controller measures its own speed and the distance to the vehicle ahead, runs a PID control algorithm, and adjusts its own speed by setting a new throttle value.

Executing the adaptive cruise control system on an embedded hardware platform, which can be prone to errors, will potentially lead to safety hazards. Evidently, a major safety hazard arises when the required distance to the vehicle in front is not maintained. Errors in the integrated hardware platform can be classified either as permanent errors, e.g., due to aging and wear effects, or as transient errors (so-called soft errors), which can be caused, for example, by particle impacts in the integrated circuits. A commonly used method to mitigate the effects of soft errors on a calculation is to use the dual modular redundancy (DMR) pattern.

Figure 2 shows an example of the DMR mechanism used on the PID controller. The DMR mechanism duplicates the function and compares the two results. If both calculations return the same result, the motor throttle is set, otherwise an error handler is triggered.

USF in the Development Flow

Organizations often have complex development workflows in place for the design of safety-critical systems, involving many tools and different input and output formats. The USF methodology aims to easily integrate with existing development workflows and tools to automate the step of generating and integrating application-specific safety mechanisms into the functional design across domains by using one pattern description per safety mechanism type. Figure 3 shows how the separate steps of the USF interact with a typical design flow. The flow can be divided into two different branches,

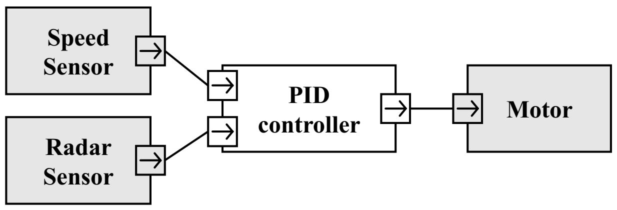
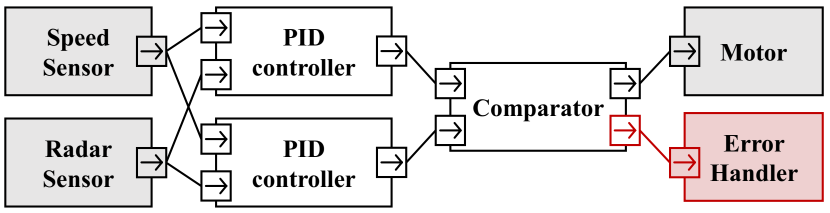


Fig. 1 Adaptive cruise control (ACC)

Fig. 2 ACC with dual modular redundancy (DMR)



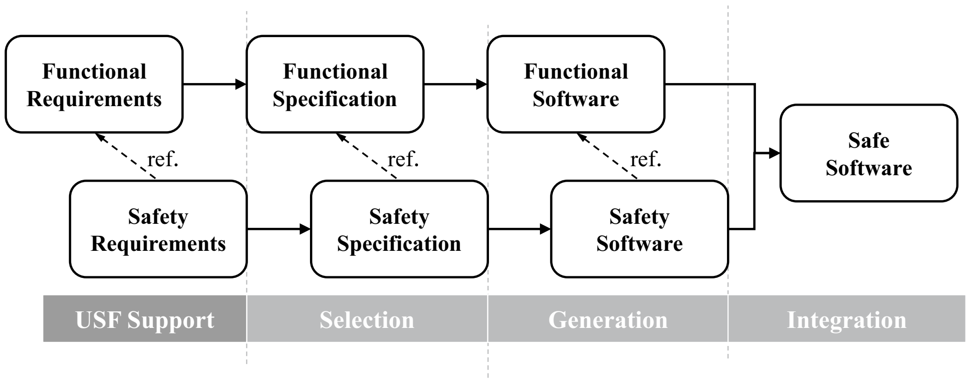


Fig. 3 Development flow with USF support

functional and safety-related development, which result in a combined safe software.

In general, function development follows the familiar three steps: elaboration of functional requirements, creation of the corresponding specification, and finally implementing the functional software. Based on the functional requirements and a safety analysis of the system, the safety requirements are derived. Analogous to functional development, a safety specification is then drafted based on the safety requirements and the functional specification, which includes the necessary software safety mechanisms. In the USF methodology, this is achieved by selecting and configuring appropriate safety mechanisms based on predefined safety patterns from a library and linking them to the functional design. This simplifies not only the creation of the safety specification, but also enables a fully formalized one.

The safety specification can then be used to implement the safety mechanisms fully automatically, i.e., to generate the safety software and then to incorporate the generated safety software into the functional software, thus resulting in the final safe software. To facilitate the creation of safety mechanisms, the core of the USF contains the following two parts:

1. USF metamodel: A domain-independent metamodel used to express the structure of functionality, including the data and control flow.
2. UTL: A transformation language for specifying patterns of safety mechanisms and their integration into USF-based models.

Safety patterns consist of a definition including the configuration parameters defined in the USF metamodel and a transformation script written in UTL that specifies the changes to the functional design. These patterns are collected in a library, which can be easily extended during development by adding new patterns. The safety pattern can then be applied in a pure USF model by specifying components within the model that should be protected by the given mechanism, and eventually executing the transformation.

The process of applying a DMR pattern to a USF element is illustrated in Fig. 4. An in-depth explanation of the metamodel is provided in Section "USF Metamodel".

To truly take advantage of the USF, safety mechanisms have to be applied to a domain-specific model instead of a USF-based model. For this, the UTL scripts are interpreted for the context of that domain, which can be automated by integrating USF support into a domain-specific tool. There are numerous implementation options to achieve this, and the optimal solution depends on the domain and the existing tool infrastructure. A detailed description of the UTL and different options to integrate them into a domain-specific tool is given in Section "USF Transformation Language". But in general, they implement the following functions:

- A mapping between domain elements and abstract USF elements.
- A process for annotating and configuring the safety pattern to model elements using the mapping.

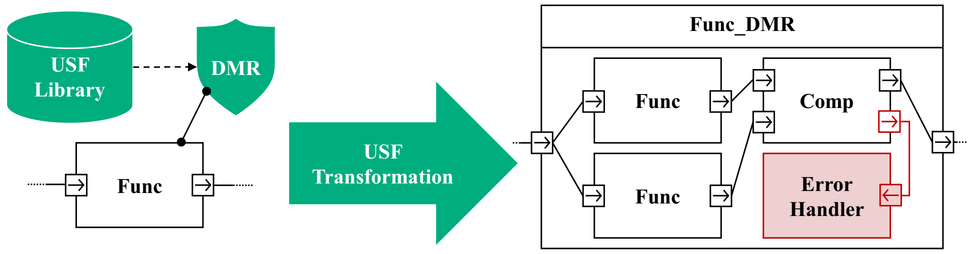


Fig. 4 Application of a DMR pattern in USF

- A method to restructure the domain-specific code according to the transformations.
- An approach for generating domain-specific implementations of the newly introduced subcomponents of the pattern (e.g., comparator functions, specific error handlers).

The USF can be applied not only to model-based approaches, but also to conventional programming languages. An example for the application of a DMR mechanism for a section of C Code is given in Fig. 5. The C code is transformed by executing all transformation steps from the UTL script in the C domain (dark green), analogous to the transformation in the USF domain (light green). A detailed description of the USF, as well as how to apply the USF to specific domains using tools, is described in the following sections.

USF Metamodel

The USF metamodel was developed to make sure that all tools are based on the same concepts, and it is therefore the foundation for a comprehensive tool support. The USF model was inspired by well-known modeling languages like UML and SysML to enable low-threshold entry for modeling experts. It also targets simplicity, which is often a requirement in the safety domain. The main concepts of the USF metamodel are outlined in this section. The full USF metamodel as well as supplementary materials are available at https://www.universalsafetyformat.org/.

Structured Elements and Flows

Blocks, ports, and connections are provided in the USF metamodel. These concepts can be used to describe the system structure and the dataflow of the functional model. The

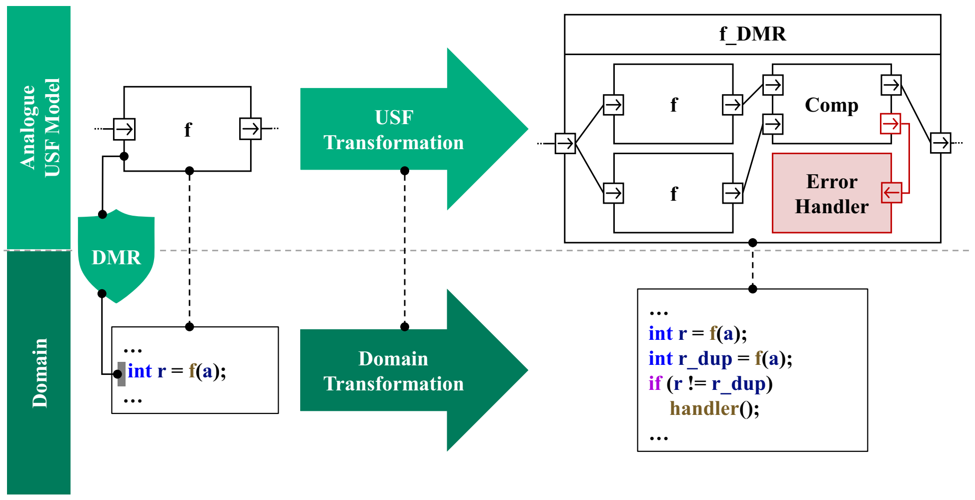
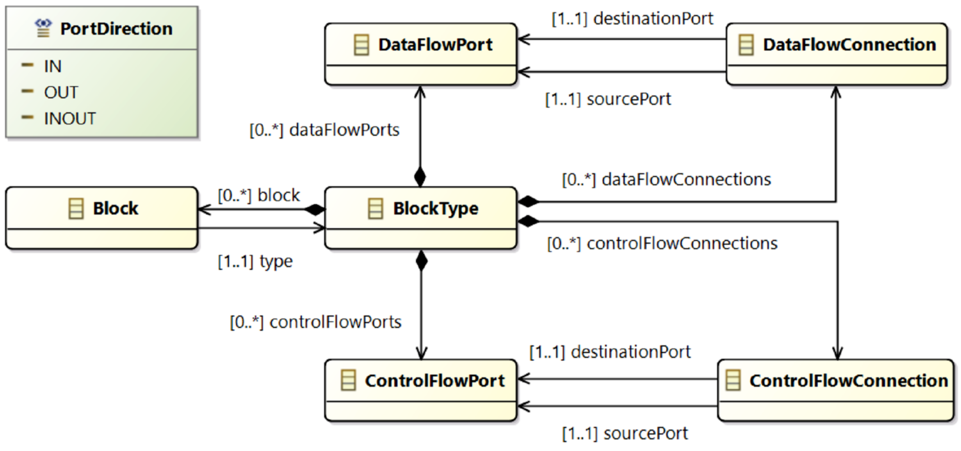


Fig. 5 Application of a DMR pattern in USF and C

Fig. 6 Blocks, ports, and connections [11]



application of the USF illustrated that for a proper modeling of some patterns control flow has to be taken into account as well. In order to address this, the metamodel was enhanced by dedicated ports and connections to specify control flow. Figure 6 shows the part of the metamodel to describe dataflow and control flow in one model.

A `Block`, characterized by a `BlockType`, is used to model the functionality of a system. The information flow between blocks is described by a `DataFlowConnection`, which connects two `DataFlowPorts`. A `DataFlowPort` defines an interface of a block and can be typed by the USF type concept. The `PortDirection` specifies the direction of a port, e.g., IN or OUT. A type concept is also part of the USF metamodel and provides elements like `StructType`, `ArrayType`, `EnumerationType`, `TemplateType`, and `PrimitiveType`. This provides the capabilities to type ports.

`ControlFlowPort` and `ControlFlowConnection` are provided to describe control flows. For a proper control flow modeling additional concepts shown in Fig. 7 are provided. There are three options to split a control flow: decisions, synchronous, and asynchronous control flow. Decisions are typically if-else branches, which allow different control flows based on conditions. `DecisionNode` and `MergeNode` are part of the metamodel to describe this. Synchronous control flow is usually used to represent the parallel work in multiple tasks. A `ForkNode` is used to split the control flow in more than one paths, while a `JoinNode` synchronizes the control flows and joins all paths back into one. To model asynchronous control flow or the reaction on certain signals, the `SendSignalNode` and `ReceiveSignalNode` can be used. They are intended to react on signals sent by other elements or even external libraries and hardware.

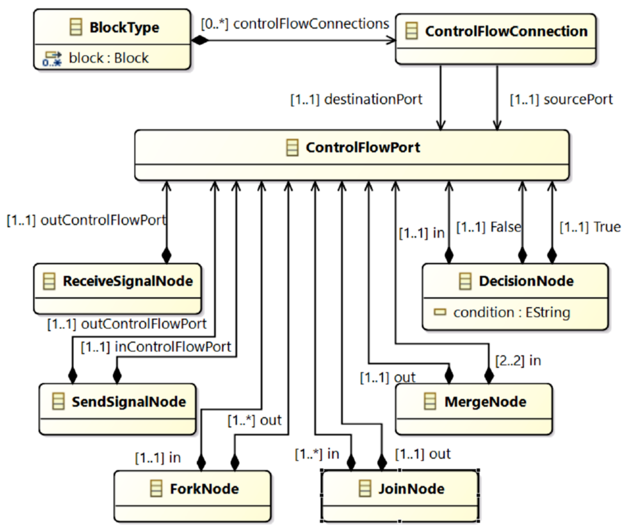


Fig. 7 Control flow concepts

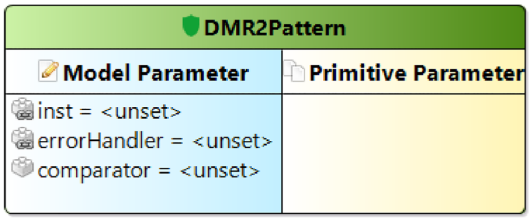
Safety Pattern

Safety patterns are formalized specifications for safety mechanisms, which can be seen as technical solutions to protect a functional system. The USF metamodel provides the capabilities to define the interface of a safety pattern. This interface definition shows all the required parameters and therefore provides a brief documentation. An example of a safety pattern definition is shown in Fig. 8.

To apply a safety pattern, a model transformation is executed. Such a transformation is implemented in UTL, a transformation language described in more detail later in this paper. A skeleton of a transformation script can be created from the safety pattern definition, listing all the parameters, and providing a basic validation of the parameters. The pattern specification as well as its assignment provide the input for the transformation. The functional model is converted by the model transformation into an enriched model where all assigned safety patterns have been applied. The main USF concepts for safety patterns are shown in Fig. 9.

All the required parameters that are needed for the transformation are specified in the model by a `SafetyPattern`, which can be seen as a template that needs to be configured to run the transformation. A `SafetyPatternApplication` is assigned to system elements, like blocks and connections, to apply a `SafetyPattern`. To run the transformation properly, the template with all the defined parameters needs to be filled in with concrete values to configure the transformation. Values can be references to model elements or primitive values.

Several properties allow to add more details to SafetyPattern descriptions, like adding traceability to safety goals or requirements. This provides the capabilities to describe to which extent a pattern may support fulfilling specific safety standard requirements or recommendations.


DMR2Pattern

Model Parameter	Primitive Parameter
inst = <unset>	
errorHandler = <unset>	
comparator = <unset>	

Fig. 8 Safety pattern definition [11]

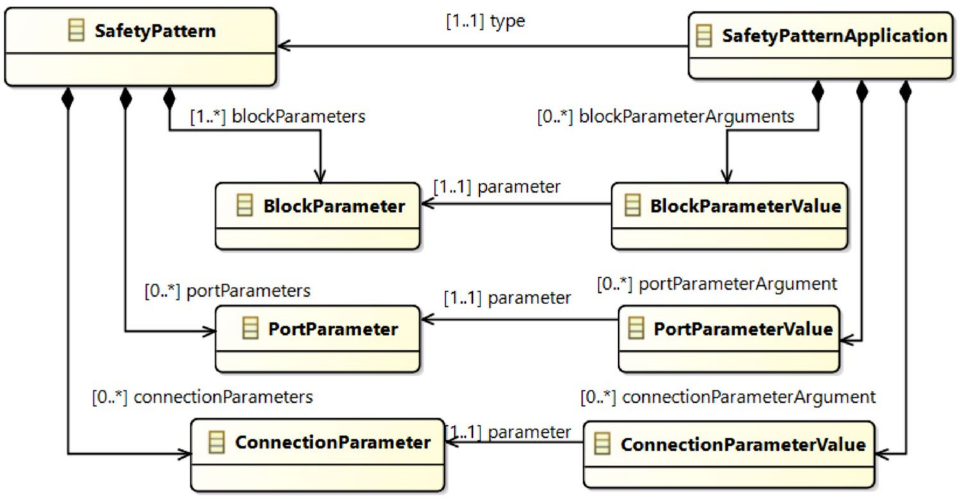


Fig. 9 Safety pattern and safety pattern application [11]

Pattern Application

This section describes how the USF can be used to apply safety mechanisms to a functional model of a system. The first step is to define the `SafetyPattern`. Figure 8 shows the definition of the DMR pattern in a safety pattern diagram.

On the left side, all required model parameters of a `SafetyPattern` are listed in the blue box. These parameters need to be instantiated during the pattern application by references to concrete model elements. The yellow box shows additional primitive parameters, which usually consist of boolean, integer and string values. These primitive parameters can have default values, which can be overwritten in the safety pattern application.

Figure 10 shows the functional model of an ACC system in a block diagram and how the DMR pattern is applied for the `Controller` task. The green shield symbol shows an instance of the `DMR2Pattern` with a list of all parameters according to the pattern definition. All model parameters are applied by drawing assignment links to model elements in the diagram. The `inst` parameter is assigned to the `Controller` block, since this is the block that should be duplicated for a redundant execution. The `DMR Comparator` is a new block that is needed to compare the results of the redundant execution. This comparator is added automatically to the block diagram by the UTL script as well as the redundant execution of the `Controller`. The assignment of the `ErrorHandler` shows how an existing error handler can be used as well. These assignments are passed to the transformation script when the transformation is executed.

Note that USF does not offer rule-based application of safety patterns using pattern matching. This is a deliberate restriction, as there might be very similar model contexts where the decision of a safety engineer on the actual application of a safety pattern has to be based on semantic information, which cannot be retrieved from the model automatically. However, if the domain-specific model provides all necessary information, a generic algorithm for rule-based

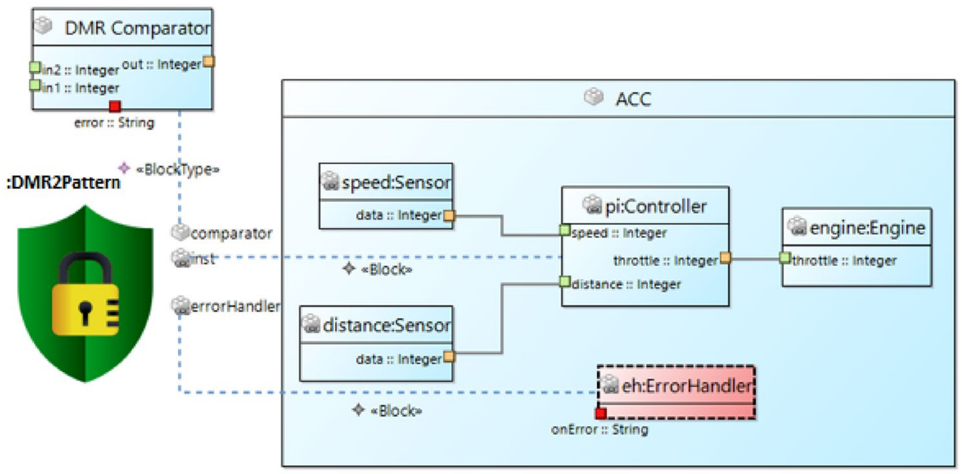


Fig. 10 Safety pattern application [11]

Table 1 Short list of used stereotypes

Stereotype	Description
≪ HW ≫	General hardware resource
≪ CPU ≫	CPU resource
≪ IRQ ≫	Interrupt request

pattern application can be added on top of USF (e.g., apply DMR to all C-functions with at least one GPIO access).

Hardware Abstraction Layer

Some safety patterns require the use of software as well as hardware resources. Therefore, a general extension mechanism was added to the USF. Such extension mechanisms are well known from modeling languages like UML. Stereotypes and parameters can be used to extend the vocabulary of USF and can be assigned to `BlockTypes`, `Blocks`, `Ports`, and `Connections`. Table 1 provides a list of stereotypes that have been used to describe hardware resources in the ACC example. This list can be easily extended for project specific needs by defining additional stereotypes.

Two additional relationships have been added to USF to describe the usage of hardware resources. Figure 11 shows these relationships and how they are used.

The `deployedAt` relationship describes that a `Block` is executed on a specific target, e.g., a timer is executed on a specific CPU. The `uses` relationship describes that a `Block` uses a specific hardware resource. Both relationships can used by the code generator to create the target specific code.

USF Transformation Language

In the previous section, we introduced the metamodel for USF. In order to reach our primary goal of automatically integrating safety mechanisms into code and models, the first step is to formally define these artifacts as instances of the USF metamodel. The actual *weaving* of patterns into such models can then be defined as *model transformations*. In the following text, we will use the term *weaving* for the automatic integration of safety mechanisms, following the terminology of aspect-oriented programming (AOP [13]).

Although it is possible to implement model transformations using general-purpose languages like Java or C++, it has turned out that domain-specific languages are a better approach for this task. These model transformation DSLs can support typical model traversal and manipulation operations as first-class language concepts, thus allowing a more concise and effective specification of transformations. In order to understand the features to be supported by a USF

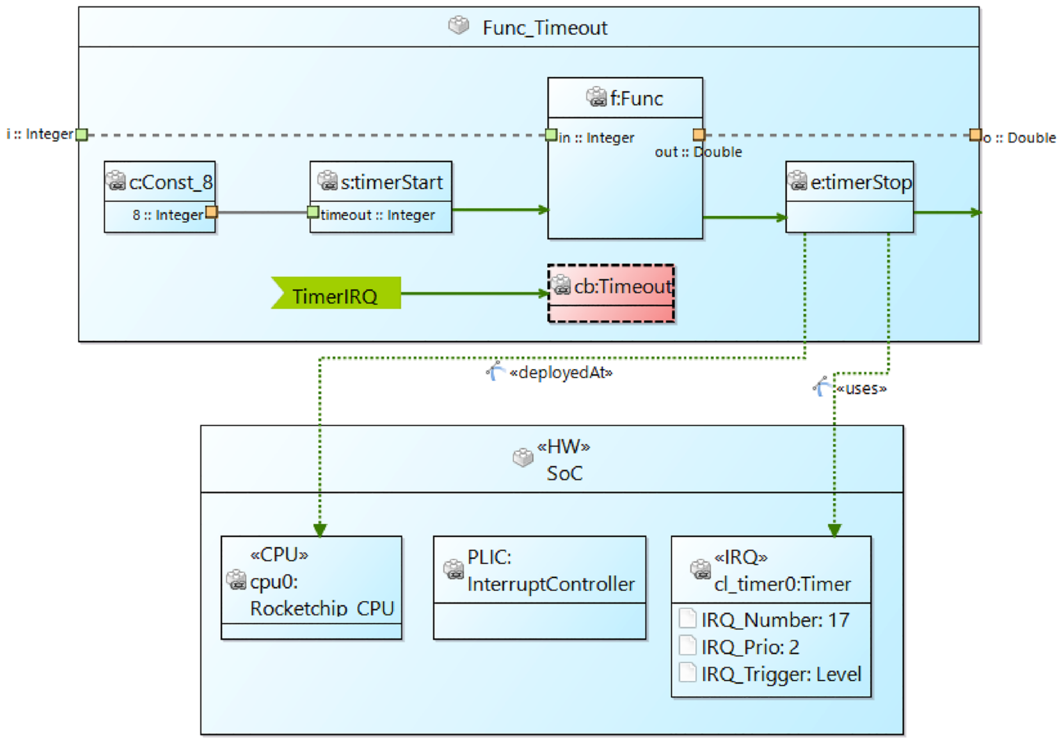


Fig. 11 `deployedAt` and `uses` relationships

transformation language, we will review the requirements specific to safety weaving in the next section.

Requirements for the USF Transformation Language

The specification of model transformations is a mature field. Therefore, a couple of general-purpose transformation languages exist (e.g., ATL [14]), providing different degrees of declarative vs. imperative representation of the model transformations. The following list contains our requirements for the USF transformation language, marked with one or two stars if they are specific (*) or highly specific (**) to our goal of safety weaving and cannot be supported directly by general-purpose transformation languages.

- The language should be easy to use esp. for safety engineers, which usually do not have a strong background on software development. (**)
- The transformation language should allow the definition of safety patterns independent from the target domain. E.g., it should be possible to apply the same model transformation to C code as well as to SysML models. (*)
- As the model transformations will be done from a source model to a target model on the same USF metamodel, the language has to support uni-directional, endogenous transformation specifications only. Bi-directional transformations or transformations between two different metamodels are not needed. (*)
- The language should be statically typed in order to avoid runtime errors during transformation application.
- The language shall support basic data types (esp. integers, strings and booleans) and the usual operations (e.g., string concatenation) on these types.
- It also has to support the canonical set of operations on all concepts from the USF metamodel.
- For the sake of simplicity, the language should contain a basic set of generic operations and control structures, but not more. (*)
- It should be possible to modularize model transformations, i.e., build complex transformations using more basic ones as building blocks. (*)
- The language should allow convenient creation of USF model fragments (using models as blueprint, e.g., construction via quotation techniques). (**)
- The language shall support the integration with glue code fragments specific to concepts from the target domain. E.g., the C code fragment for triggering a hardware interrupt on a specific hardware platform should be seamlessly linked to the domain-independent model transformation. (**)

This analysis shows that many of our requirements cannot be met sufficiently by general-purpose transformation languages. Thus, in the following we propose UTL (USF Transformation Language), which is a transformation language specific for safety weaving based on USF.

UTL is designed to be realized using a convenient textual concrete syntax, implemented by a state-of-the-art parser like ANTLR. Additionally, it can be enriched with more elaborate notational elements if the tool platform allows it (e.g., the language workbench tool MPS [15] by JetBrains with its projectional editing approach).

Language Concepts of UTL

This section gives an overview on the various concepts of the UTL language. In general, our design guideline was to feature a primary imperative language style with some functional additions, as users with limited experience in software development often can grasp this style better than a purely functional one (cf. the popularity of scripting languages like Python in the ML and scientific communities).

Statically Typed Expression Language

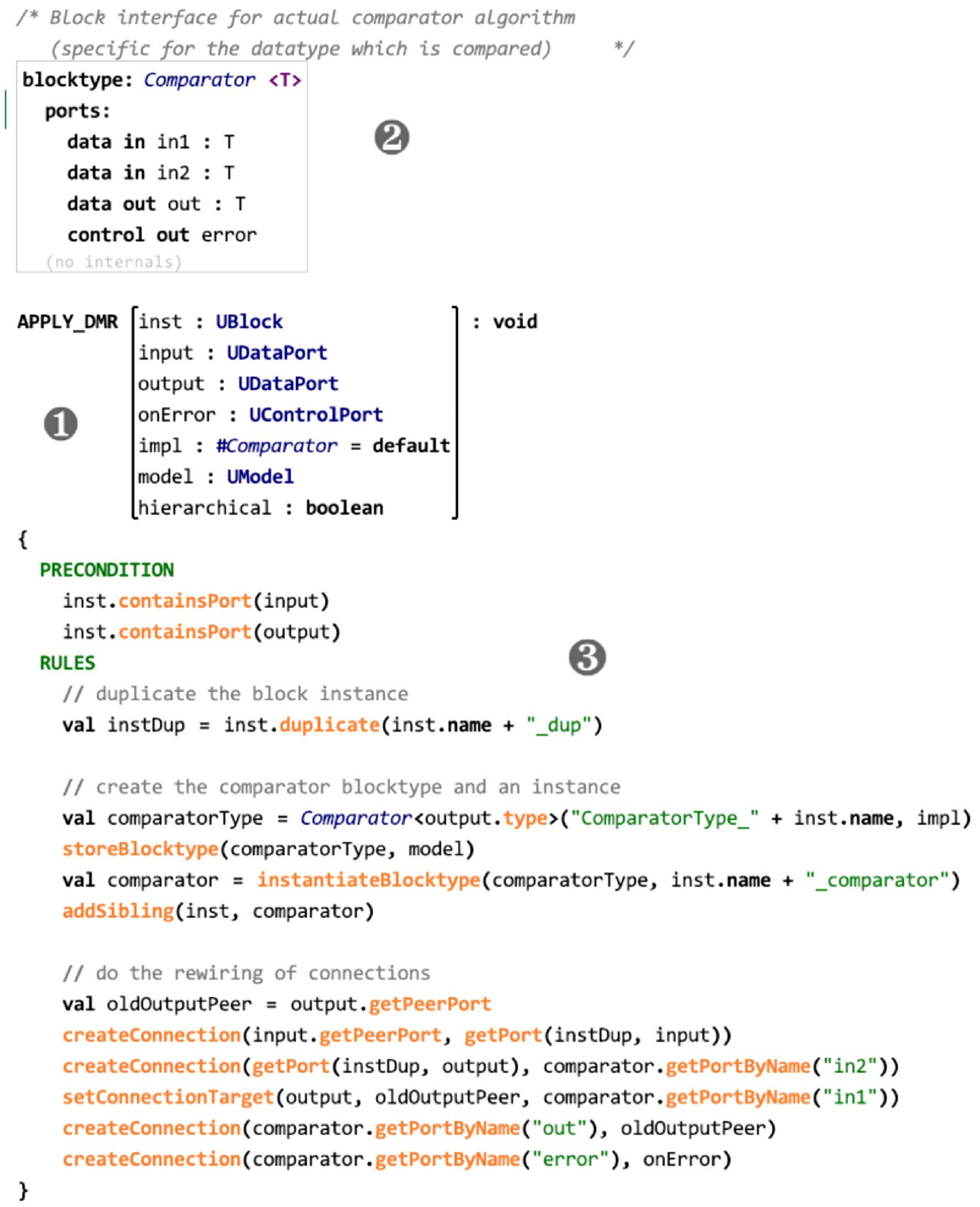
UTL is a statically typed language with type inference. The basis of the UTL type system is a canonical set of primitive types (i.e., boolean, integer, string) and operations on these types, e.g., string concatenation. In addition, a subset of the USF metamodel concepts is available as types in UTL, esp. `Block`, `DataPort` and `ControlPort`. Non-mutable local variables can be defined for convenience to increase code readability. Moreover, mutable variables are available to support the imperative language style.

Modular Transformations

In order to support the requirement of building complex transformations from smaller ones, UTL adopts the paradigm of function calls. I.e., the interface of each model transformation is defined by a signature consisting of a name, a set of named parameters with types and a return type (see item ➊ in Fig. 12). Using this signature, a transformation can be "called" from annotations in the domain model or in the USF model. Usually, annotations will need to refer to specific domain elements, which corresponds to code locations if the target domain is program code. These model references can be considered as *join points*, which is a term for code locations from aspect-orientated programming (AOP [13]).

Operation API

The application programming interface (API) of UTL consists of a large set of operations on USF model elements. These can be divided into two groups: The set of canonical operations defined by the USF metamodel



```
/* Block interface for actual comparator algorithm
   (specific for the datatype which is compared)     */
blocktype: Comparator <T>
  ports:
    data in in1 : T
    data in in2 : T
    data out out : T
    control out error
  (no internals)

APPLY_DMR [inst : UBlock                  ] : void
          [input : UDataPort              ]
          [output : UDataPort             ]
          [onError : UControlPort         ]
          [impl : #Comparator = default   ]
          [model : UModel                 ]
          [hierarchical : boolean         ]
{
  PRECONDITION
    inst.containsPort(input)
    inst.containsPort(output)
  RULES
    // duplicate the block instance
    val instDup = inst.duplicate(inst.name + "_dup")

    // create the comparator blocktype and an instance
    val comparatorType = Comparator<output.type>("ComparatorType_" + inst.name, impl)
    storeBlocktype(comparatorType, model)
    val comparator = instantiateBlocktype(comparatorType, inst.name + "_comparator")
    addSibling(inst, comparator)

    // do the rewiring of connections
    val oldOutputPeer = output.getPeerPort
    createConnection(input.getPeerPort, getPort(instDup, input))
    createConnection(getPort(instDup, output), comparator.getPortByName("in2"))
    setConnectionTarget(output, oldOutputPeer, comparator.getPortByName("in1"))
    createConnection(comparator.getPortByName("out"), oldOutputPeer)
    createConnection(comparator.getPortByName("error"), onError)
}
```

Fig. 12 UTL example transformation: DMR mechanism [11]

(e.g., getters and setters for attributes), and a set of helper operations which provide additional logic or shortcuts for typical patterns (e.g., `createConnection()` for creating new connection elements and linking them to the proper `Port` nodes).

Creation of USF Model Fragments

A common pattern in UTL model transformations is the necessity to create USF model fragments. In order to provide a convenient means to define these fragments and

instantiate them in the target model, transformation signatures might also include block type definitions (according to the USF metamodel). E.g., the transformation depicted at item
➋ in Fig. 12 uses the block type `Comparator`< T >.

The complementary functionality for these block type definition parameters is the creation of actual model fragments, using the parameter value as a blueprint. For this, UTL supports a *constructor* syntax. A constructor call creates a new block type, using an existing one as a blueprint. This is shown for the `Comparator` block type in the example (item
➌ in Fig. 12). The blueprint `Comparator` can be defined using any USF model editor. In the example, it has been defined using a textual syntax as part of the transformation signature (item
➋). The blueprint is passed to the transformation body as a parameter `impl`. The type of this parameter is defined by a special hashtag-syntax; the `=default` syntax specifies that the parameter is optional. The first parameter of the constructor is the name of the new block type. With the second parameter, a domain-specific implementation of type `Comparator`< T > can be provided.

Abstracting from Domain-Specific Details

The UTL language (like the USF metamodel) is not restricted to a specific target domain. However, in order to apply transformations on models of a given target domain it is required to inject domain-specific details, e.g., C glue code. The memento-like pattern based on `#Comparator` and the constructor syntax can be used to inject domain-specific behavior as implementation of the created block type. E.g., for the C domain this can be a C code snippet, which adheres to the interface defined by the block type's ports. The specific value of the `#Comparator` parameter will be initialized as part of the annotation in the domain model (e.g., the C code) and is "tunneled" through the transformation script until the constructor executes.

Aspects of Executing UTL Transformations

By using UTL, safety mechanisms can be specified independently from specific target domains. This is accomplished by "implementing" the safety mechanism as a (mostly imperative) transformation script. For the automatic weaving process, UTL scripts have to be applied to input models from a given target domain (e.g., C code). This section describes the interplay of the different artifacts being used throughout this process. We will first discuss the options for executing UTL model transformations, and afterwards describe the commonalities and differences of weaving for structural models and program code.

Transformation Execution Approaches

For the execution of transformation scripts, which are implemented using UTL, there are basically three different options:

- Option 1: Translating each input domain-model into a USF model and executing the transformation in a generic way. After the execution, the result model must be translated back to the target domain.
- Option 2: Translating the transformation script itself into general-purpose code (e.g., Java), which can then be executed directly on a representation of the domain-specific model.
- Option 3: Using an interpreter for UTL with a domain-specific backend, mapping each UTL operation to a domain-specific implementation.

Option 1 has the advantage that the actual execution of UTL has to be implemented only once (for USF). For each target domain, just the implementation of two mapping transformations from the domain model to USF and back have to be provided. One major disadvantage of option 1 is that the whole input model has to be sent through the three-step pipeline, as it cannot be known in advance where the actual weaving will happen. Option 1 is being used by the Eclipse-based safety toolchain for Simulink models as described in Section "Simulink".

Option 2 requires the implementation of a code generator for UTL scripts, as these scripts have to be converted to some general-purpose language. As with many other code generation techniques, the generated transformation code has to be supported by a proper runtime library.

Option 3 is similar to option 2, as each UTL language concept has to be executed by a piece of general-purpose code which implements the UTL operation semantics on a given target domain. However, this is easier than with option 2, as each operation can be tackled separately, and significant parts of the runtime can be domain-agnostic. The additional benefit of the interpreter approach of option 3 is that only those parts of the target domain model have to be manipulated which are subject to weaving. Option 3 is being used by the C-code weaving engine of the SafetyWeaver tool (see second part of Section "Tool support").

When selecting the best option for the implementation of a UTL transformation engine, a variety of factors have to be taken into account: Constraints imposed by the tool platform and the available technologies, integration with other tools as part of a toolchain, the complexity of the target domain, and requirements due to tool qualification in a safety-related context. Esp. for program code domains (e.g., C code) the mapping to USF concepts might be complex.

Weaving for Structural Models

The target domain of structural models allows representing hierarchically structured architectures with components and ports. Typical industry-relevant domains from this category are AUTOSAR, Simulink, SysML, and other SysML-like proprietary models. For instance, applications in the automotive domain often use the Eclipse platform *Artop* (an EMF-based AUTOSAR implementation [16]). This can be integrated easily with the USF reference implementation, as both frameworks use the EMF technology for the definition of their metamodels. Often the programming language *Xtend* [17] is being used on this technology stack for implementing transformation engines.

The mapping from structural models to USF is quite straightforward. USF block types and blocks will represent components in the target domain, and port concepts can often be mapped directly. Component hierarchies (consisting of nested components and subcomponents) are natively supported by USF blocks and block types as well. Therefore, the execution of UTL transformation scripts on these structures is possible with a minimal additional mapping logic.

Weaving for Program Code, esp. C

Our implementation of weaving tools showed that C code safety weaving is more challenging than weaving for structural models (e.g., SysML block diagrams). The artifacts required for C code weaving and their interplay are shown in Fig. 13.

In compilers and other code-related tools, program code (e.g., C) is represented as abstract syntax tree (AST). In order to map this domain to USF, elements of the AST have to be represented by USF blocks and other elements. USF has been designed to cover this, esp. by supporting dataflow

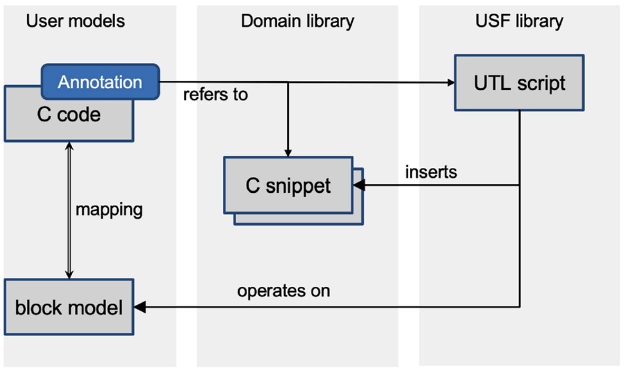


Fig. 13 Applying transformations (example: C domain) [11]

Table 2 Mapping of USF to C code elements (subset)

USF port type	C code element
Dataflow output port	New local variable with initializer
Dataflow input port	Read access to local variable
Control flow output port	Sequential execution or C-goto
Control flow input port	Sequential execution or C-label

and control flow concepts. Despite this support, the mapping between a C AST and USF block models is not straightforward. E.g., the conceptual mappings for some elements of the C AST and USF ports are shown in Table 2.

In our reference implementation using the JetBrains MPS language workbench, a UTL interpreter with an API for domain-specific plug-ins has been realized. The C domain plug-in creates the USF model from the input C code on the fly, starting from the annotated code elements (e.g., C functions or C blocks). The UTL interpreter will create new blocks and connections depending on the statements of the actual transformation scripts. In a post-processing step, the resulting USF model is converted into C AST elements and manifested as code. The control flow connections on USF block level determine the order of the newly created C code blocks. The post-processing also uses a rule-based approach to replace goto/label pairs (introduced during the weaving) by structured code.

The domain library (cf. Fig. 13) contains glue code fragments (C snippets) necessary for linking the code resulting from the weaving with the actual capabilities and APIs of the target system. E.g., initializing a hardware timer might be done differently for specific combinations of firmware and processor hardware. This approach allows assigning different responsibilities to corresponding experts with matching skills, e.g., an embedded engineer owns the domain library, whereas the safety expert takes care of the USF library.

Evaluation

Several demonstrators have been developed and assessed to validate the presented approach. In these demonstrators, various safety patterns are used at different levels of abstraction. Starting point in all demonstrators is the functional software system, which is provided as C source code, Simulink, or SysML models. Common safety patterns can be organized in libraries and provided to the end users. This allows an out of the box usage of proven safety patterns for any functional model. Some safety patterns that have been used in the demonstrators are listed in Table 3. In this paper the previously described ACC system is used as an example to describe how safety patterns can be described and applied.

Table 3 Selected safety patterns supported by USF and their mention in safety standards

Pattern name	Description	IEC 61508	ISO 26262
DMR	Dual modular redundancy	✓	
TMR	Triple modular redundancy	✓	
Watchdog	Hardware watchdog	✓	✓
CRC	CRC generation and checks	✓	✓
ESM-ICU	External safety mechanism: interrupt controller unit test		

Tool Support

The application of safety patterns and therefore the creation of safe software systems can be done in a semi-automated way. Dedicated USF modeling and model transformation tools have been developed to provide an appropriate tool support for the description and application of safety patterns. This section gives a brief overview of the tools *SafetyModeler* and *SafetyWeaver*. Since the development of the demonstrators and the USF tools happened in parallel, a permanent feedback loop in an agile approach was used.

As USF and UTL are standardized specifications, it is possible for any vendor to provide an implementation using their preferred tool platform. For the Eclipse platform, SafetyModeler provides an open-source, ready-to-use implementation of both specifications. On the other hand, SafetyWeaver uses JetBrains MPS to implement the specifications. Based on the specified metamodels, exchange of USF models and UTL scripts between tools is easily possible, allowing to form tool chains for specific tasks (e.g., as demonstrated in Section "Simulink").

SafetyModeler

In order to view and to create USF models in a graphical way, a newly developed tool named SafetyModeler offers functional block modeling with dataflows and control flows. To deploy safety patterns, it also provides functionality to define safety patterns and to apply them in the functional model. Modeling capabilities can be extended by defining stereotypes and data types.

An XMI interface allows the import of functional models from other sources. If necessary, layout algorithms support the creation of diagrams in a semi-automated way. SafetyModeler provides the following main functionality:

- Block Diagrams to describe functional systems in a graphical way
- Description of dataflow and control flow in the same view
- Definition of safety patterns and safety pattern applications
- Definition of transformations for safety patterns in UTL
- Execution of transformations and visualization of the resulting model

The user interface of SafetyModeler provides several views. The main part is the drawing canvas in the middle including the symbol palettes, which allows the graphical modeling of a functional software, the safety patterns, and the safety pattern application. Figure 14 shows the block diagram of the previously described ACC system in SafetyModeler. A tree view on the left-hand side helps to easily navigate through a more complex model. Details of a selected element in the tree or in the drawing canvas can be viewed and edited in the properties view.

UTL support is also part of SafetyModeler. It offers functionality to define transformation scripts in UTL using a language sensitive editor. A skeleton for the transformation is created the first time the UTL editor is called for a defined safety pattern. To support the editing of UTL scripts the editor provides color coding and name completion for all defined functions in the USF metamodel and also highlights errors in a UTL script. The final script can be executed within SafetyModeler. This will create a new save functional model including the applied safety patterns. To verify the correct application of safety patterns in the new model, it can be visualized in SafetyModeler again.

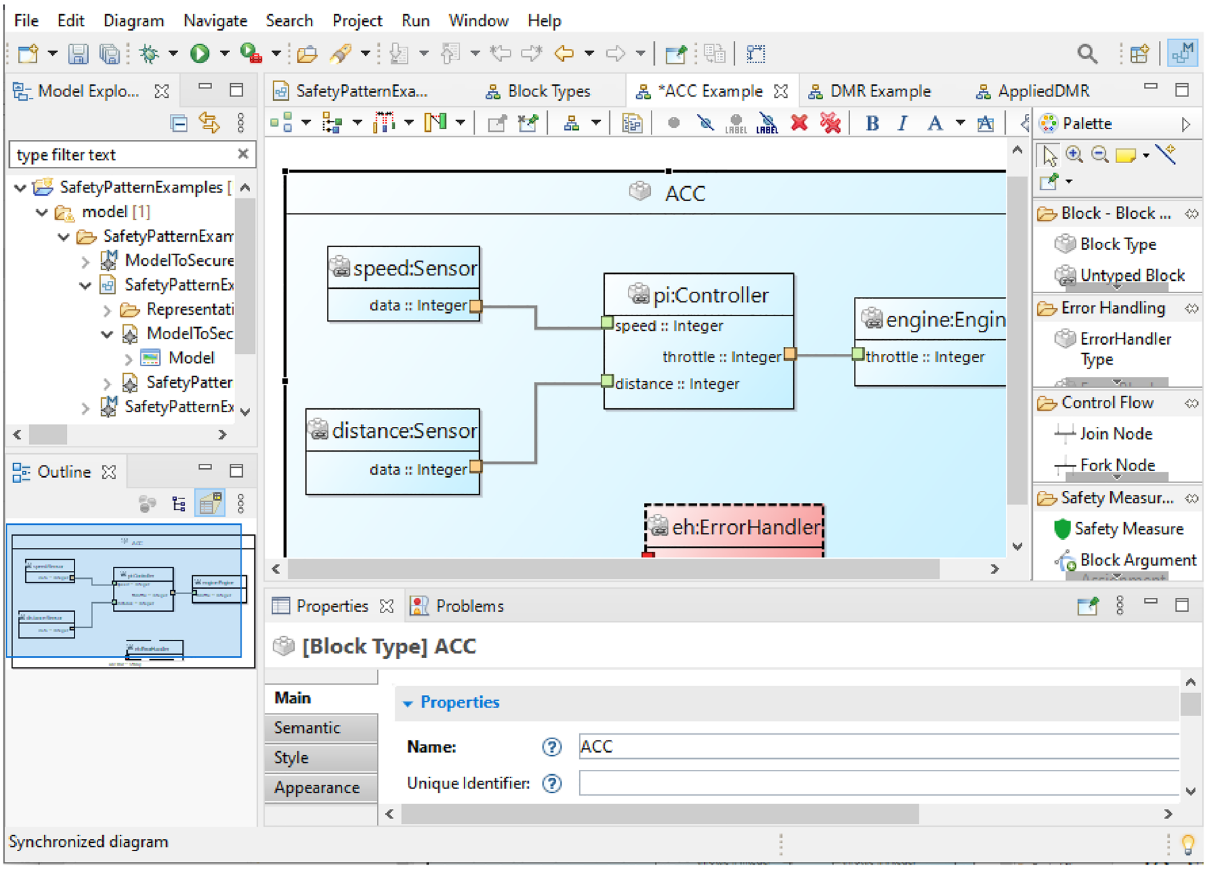
SafetyModeler was also very helpful during the development phase of the USF metamodel to instantiate and visualize certain milestones. The validation of a current USF metamodel and the identification of missing pieces was much easier. By this means, SafetyModeler supported the agile development approach for USF. It is published as open source software and available on an update site as an Eclipse plugin (cf. [12]). It can be easily installed into a running Eclipse Modeling installation.

SafetyWeaver

The *SafetyWeaver* tool mainly targets embedded developers and safety engineers. It combines the following aspects in one integrated tool environment:

- extensible IDE support for C-developers
- editor for UTL transformation scripts and C glue code
- automatic weaving for C code and other target domains
- additional features for building and using DSLs, esp. structural models

SafetyWeaver is using the JetBrains MPS platform [15], which allows combining various different notations (textual, graphical, tables, tree structures) as well as authoring new



[Block Type] ACC

Properties

Name: ACC

Unique Identifier:

Fig. 14 Views in SafetyModeler

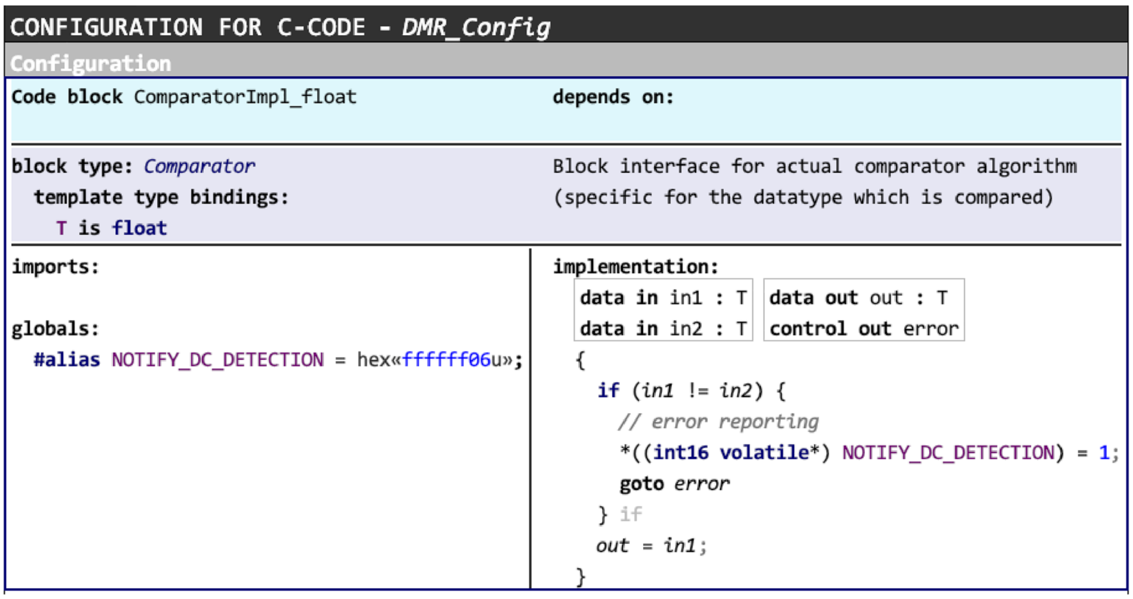
DSLs and flexibly integrating them [18]. This results in a consistent and intuitive user experience when working with SafetyWeaver. Here are some examples [11]:

- *safety engineers* can add and edit annotations directly in the C code and still get context-specific proposals (e.g., for selecting the proper transformation script and its parameters, see green elements in Figs. 20 and 22)
- *transformation authors* can provide online documentation (e.g., for each transformation and its parameters) which is presented to transformation users as type system checks and tooltips
- *platform architects* who implement the C glue code can use a code block editor which enforces the constraints defined by the transformation definition (e.g., dataflow input ports represented as read-only C variables)
- the resulting C code is automatically annotated with projected trace information, providing trace links leading back to the applied UTL-scripts, and additional glue code blocks (traceability, see Fig. 21 and Fig. 24)

Figure 15 illustrates the benefits provided by the editor for glue code: By setting the `block type` to `Comparator`, the dataflow and control flow ports of this block type uniquely define the interface for the glue code (see also Fig. 12).

E.g., the dataflow input port `in1` is represented by a read-only local variable of the same name in the `implementation`-section of the glue code editor. Note that the editor supports generic data types: For the comparator the generic type `T` will be bound to a specific type, depending on the output of the code block, which is duplicated by the DMR mechanism.

For the C support, SafetyWeaver uses the open source mbeddr platform [19]. mbeddr stores the C code as AST. SafetyWeaver relies on that representation to implement the weaving of UTL mechanisms as a direct interpretation of the UTL transformation scripts (cf. Section "Aspects of Executing UTL Transformations"). SafetyWeaver transforms only those parts of the C AST to in-memory USF models, which are required for the weaving. As part of the transformation postprocessing, the output USF model is optimized (e.g., control flow clean-up), transformed back to C AST subtrees and integrated into the original C AST. This approach allows efficient transformation even of big C codebases, as only the parts relevant for safety weaving have to be transformed.



```
CONFIGURATION FOR C-CODE - DMR_Config
Configuration
Code block ComparatorImpl_float                depends on:

block type: Comparator                         Block interface for actual comparator algorithm
  template type bindings:                      (specific for the datatype which is compared)
    T is float
imports:                                       implementation:
                                                 data in in1 : T   data out out : T
                                                 data in in2 : T   control out error
globals:                                         {
  #alias NOTIFY_DC_DETECTION = hex«ffffff06u»;     if (in1 != in2) {
                                                     // error reporting
                                                     *((int16 volatile*) NOTIFY_DC_DETECTION) = 1;
                                                     goto error
                                                   } if
                                                   out = in1;
                                                 }
```

Fig. 15 Example of the glue code editor in the SafetyWeaver tool: C code for DMR comparator

The internal implementation of UTL distinguishes between core operations and additional operations ("syntactic sugar"). SafetyWeaver reduces all syntactic sugar operations from a transformation script using the incremental Shadow Model engine [20]. This allows simplifying the actual model transformation process, as only the core language features have to be supported by the execution engine (i.e., the interpreter with all its domain-specific plug-ins). This is especially valuable because the transformations have to be applied on several different target domains.

Domain-Specific Safety Pattern Application

In this section, we demonstrate how the previously introduced tools can be used to realize domain-specific safety mechanisms.

Simulink

As noted in Section "Safety Engineering in a Nutshell", the ACC's main functionality is realized by a PID controller, which regulates the throttle according to the speed of the vehicle and the distance to the vehicle ahead in order to maintain a fixed distance between them.

Figure 16 shows an implementation of the PID controller inside a subsystem of a Simulink block diagram. Using the Simulink Embedded Coder, this model could now be used to generate C code for an embedded target. However, analogous to the introductory example, a DMR pattern is first applied to the Simulink subsystem to mitigate the effects

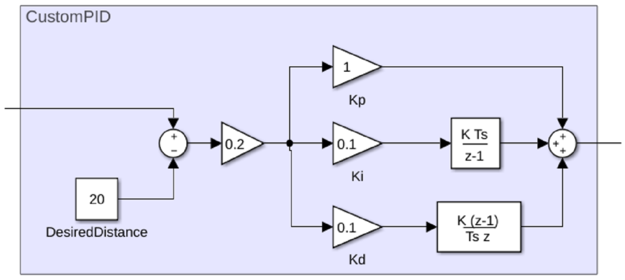


Fig. 16 PID controller in Simulink

of soft errors during computation. In order to use the USF approach with Simulink models, tool support had to be implemented. For this purpose, we decided to implement a minimal tool support that relies on SafetyModeler. As a first step, the mapping between Simulink block diagrams and the USF had to be created. Due to the structural similarity of the models, e.g., Simulink blocks to USF blocks, Simulink signals to USF data flow, etc., this was fairly straightforward. Through this mapping, we then created simple model-to-model transformations between Simulink models and USF models and vice versa with the Eclipse Epsilon plugin [21], which provides an interface to query and modify Simulink models in Eclipse via the MATLAB API.

Figure 17 shows the equivalent USF model of the PID controller, as well as an applied DMR pattern to the whole PID component in SafetyWeaver. Using the built-in transformation engine, we then applied the pattern.

To completely convert the USF model back to a Simulink model, we first had to provide Simulink implementations

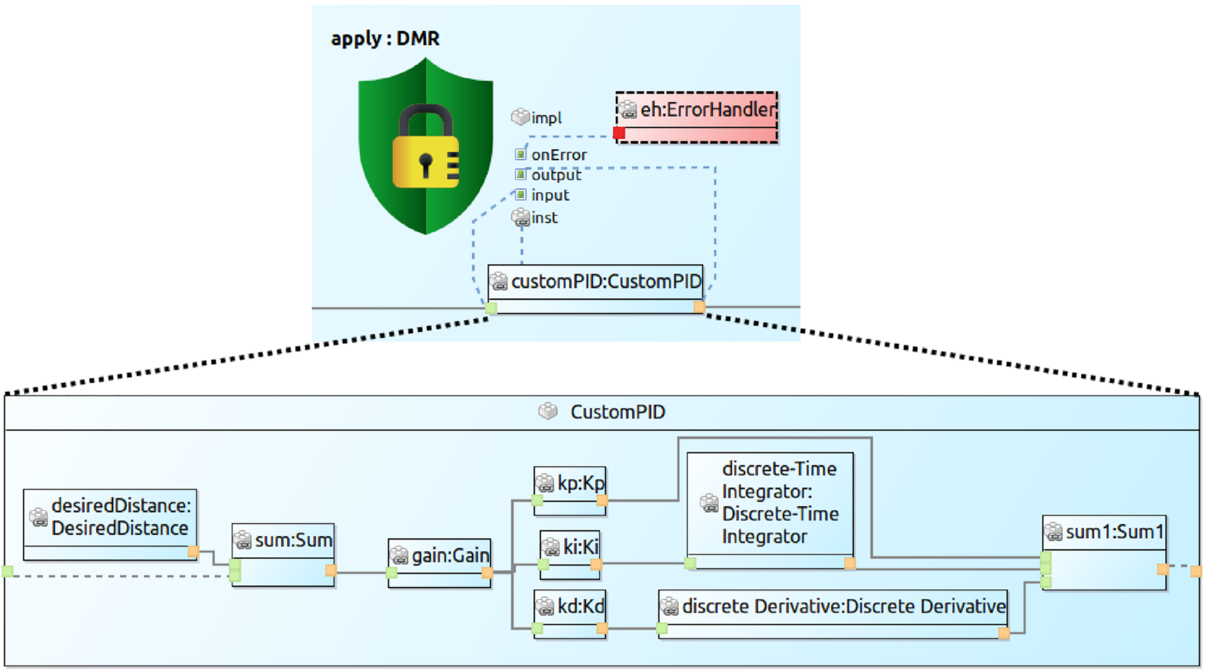

apply : DMR
impl
eh:ErrorHandler
onError
output
input
inst
customPID:CustomPID
CustomPID
desiredDistance: DesiredDistance
sum:Sum
gain:Gain
kp:Kp
ki:Ki
kd:Kd
discrete-Time Integrator: Discrete-Time Integrator
discrete Derivative:Discrete Derivative
sum1:Sum1

Fig. 17 PID controller in USF

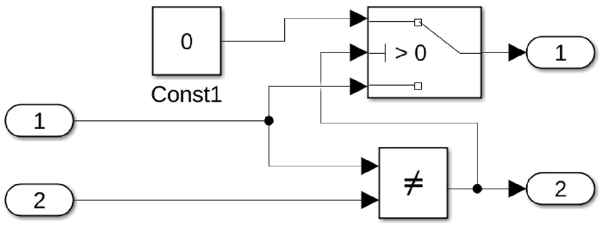


Fig. 18 Simulink implementation for the comparator

for the two new subcomponents used in the DMR pattern (comparator and error handler). To do this, we simply added a new Simulink library with the analogous subcomponents as Simulink subsystems. The implementation of the comparator can be seen in Fig. 18 and the error handler was implemented as a so-called S-function, which simply calls an existing error handler in the embedded system.

Using the transformed USF model, which includes the DMR pattern and the Simulink library, we then transformed the USF model back into a Simulink model. Figure 19 shows the result of the model transformation. The resulting Simulink subsystem was then translated with the Simulink Embedded Coder to C.

Safety-Mechanism for C

An alternative way of implementing the PID controller software for the ACC application is by using C embedded code directly. Figure 20 shows the C code for the actual PID algorithm, which has been annotated by a safety engineer in order to indicate where the DMR pattern should be applied (green box referring to the UTL transformation `APPLY_DMR`).

In the example, specifically marked locations in the C code are handed over to the transformation as parameter values. E.g., the `INPUT_V` is a join point at the local variable declaration `curr_v` (cf. Section "Language Concepts of UTL"). It represents a dataflow input port after the mapping to USF.

Figure 21 illustrates how the SafetyWeaver tool automatically changes the C code based on the input from the annotation. The green annotation box has been removed by the transformation engine and the DMR pattern has been applied instead. The data path of the manually written code has been duplicated and a new code block for the comparator aspect of the DMR mechanism has been added. The error check compares the original result variable `throttle_u` with its duplicated counterpart `throttle_u_dup` and triggers the error handling if the values are different. The glue code for error detection and reporting has been provided by a platform architect via the domain library (as explained in Section "Aspects of Executing UTL Transformations").

Another application of a very simple safety mechanism is shown in Fig. 22. Here, a *Watchdog* safety mechanism is applied in order to ensure the periodic execution of the C function `writeActuator()`. If the actuator updates

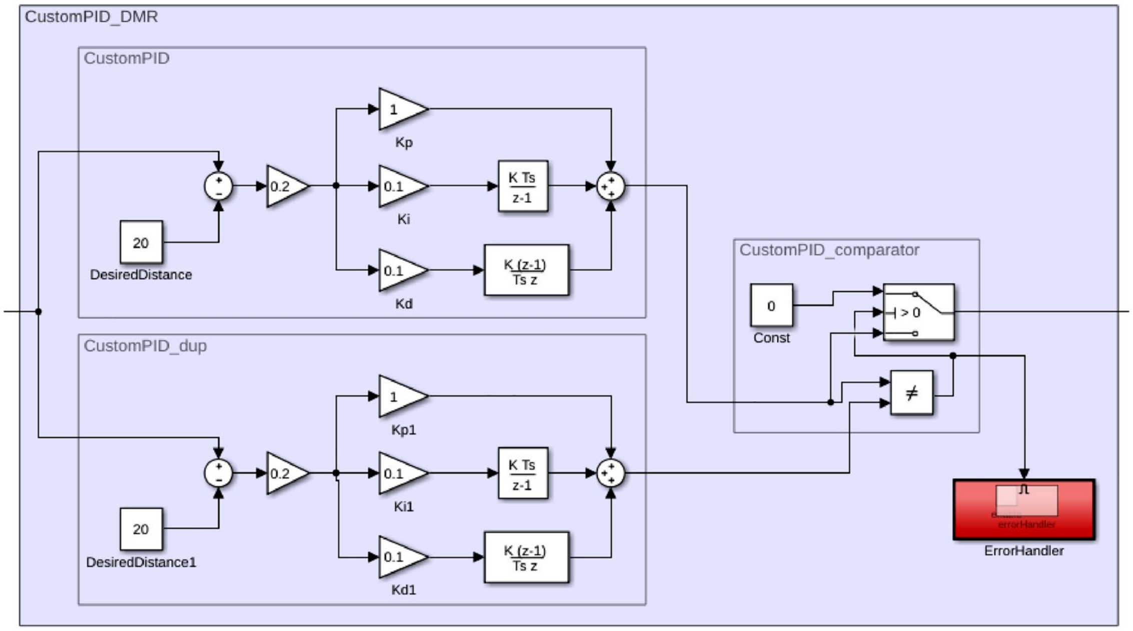
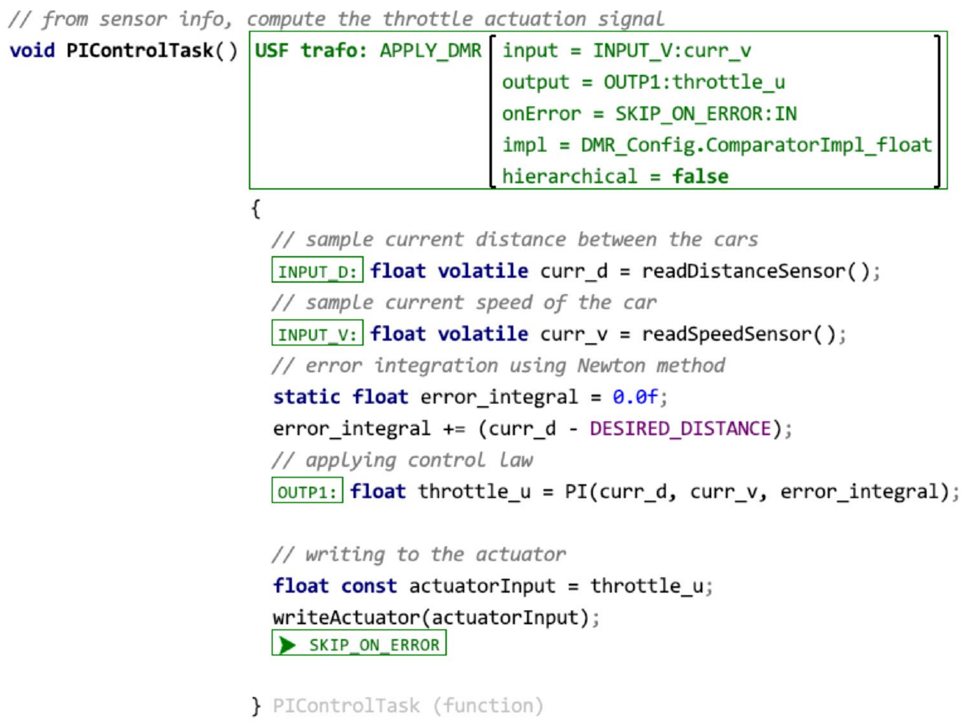


Fig. 19 PID controller with applied DMR pattern

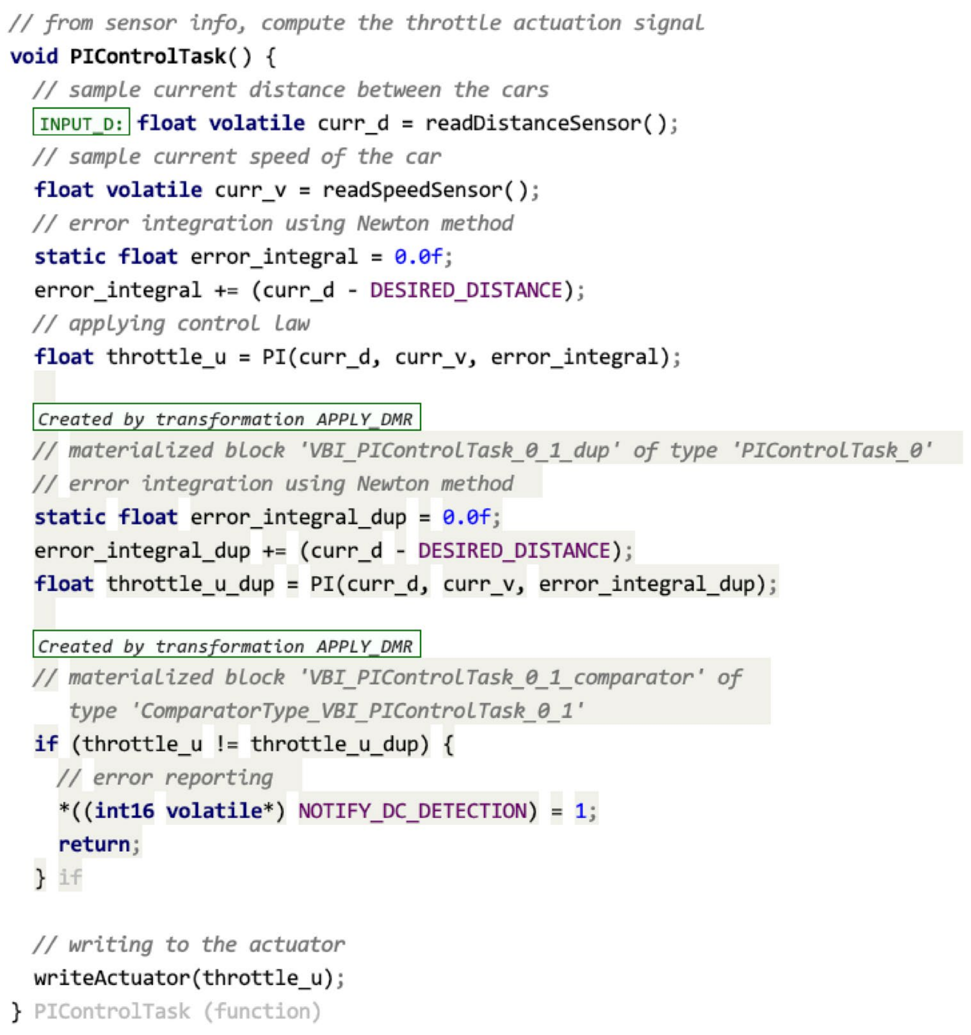
Fig. 20 Embedded code of the ACC control SW with selected DMR safety mechanism (before transformation)



```
// from sensor info, compute the throttle actuation signal
void PIControlTask() USF trafo: APPLY_DMR [ input = INPUT_V:curr_v
                                            output = OUTP1:throttle_u
                                            onError = SKIP_ON_ERROR:IN
                                            impl = DMR_Config.ComparatorImpl_float
                                            hierarchical = false                  ]
{
  // sample current distance between the cars
  INPUT_D: float volatile curr_d = readDistanceSensor();
  // sample current speed of the car
  INPUT_V: float volatile curr_v = readSpeedSensor();
  // error integration using Newton method
  static float error_integral = 0.0f;
  error_integral += (curr_d - DESIRED_DISTANCE);
  // applying control law
  OUTP1: float throttle_u = PI(curr_d, curr_v, error_integral);

  // writing to the actuator
  float const actuatorInput = throttle_u;
  writeActuator(actuatorInput);
  ▶ SKIP_ON_ERROR

} PIControlTask (function)
```

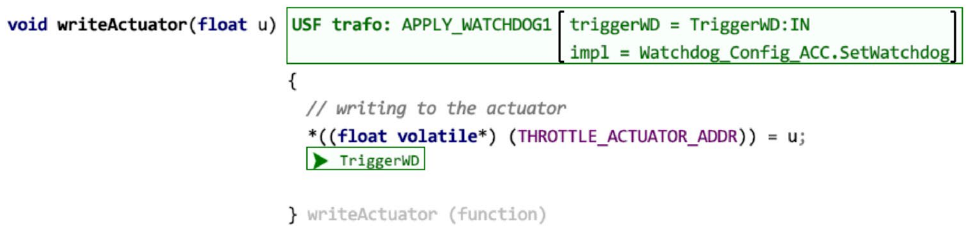
are missing at some point in time due to some hardware or software fault, an emergency situation is detected, which has to be handled properly. The only goal of the safety mechanism is triggering a hardware watchdog. If the watchdog is not triggered for a defined period of time, the error handling must kick in. The UTL transformation script is shown in Fig. 23. Its main task is to add a new code block at the location given by the control port parameter `triggerWD`.

Figure 24 illustrates the code, which has been added by the automatic safety weaving by executing the transformation script.



```
// from sensor info, compute the throttle actuation signal
void PIControlTask() {
  // sample current distance between the cars
  INPUT_D: float volatile curr_d = readDistanceSensor();
  // sample current speed of the car
  float volatile curr_v = readSpeedSensor();
  // error integration using Newton method
  static float error_integral = 0.0f;
  error_integral += (curr_d - DESIRED_DISTANCE);
  // applying control law
  float throttle_u = PI(curr_d, curr_v, error_integral);

  Created by transformation APPLY_DMR
  // materialized block 'VBI_PIControlTask_0_1_dup' of type 'PIControlTask_0'
  // error integration using Newton method
  static float error_integral_dup = 0.0f;
  error_integral_dup += (curr_d - DESIRED_DISTANCE);
  float throttle_u_dup = PI(curr_d, curr_v, error_integral_dup);

  Created by transformation APPLY_DMR
  // materialized block 'VBI_PIControlTask_0_1_comparator' of
     type 'ComparatorType_VBI_PIControlTask_0_1'
  if (throttle_u != throttle_u_dup) {
    // error reporting
    *((int16 volatile*) NOTIFY_DC_DETECTION) = 1;
    return;
  } if

  // writing to the actuator
  writeActuator(throttle_u);
} PIControlTask (function)
```

Fig. 21 Embedded code of the ACC control SW with inserted DMR safety mechanism (after transformation)



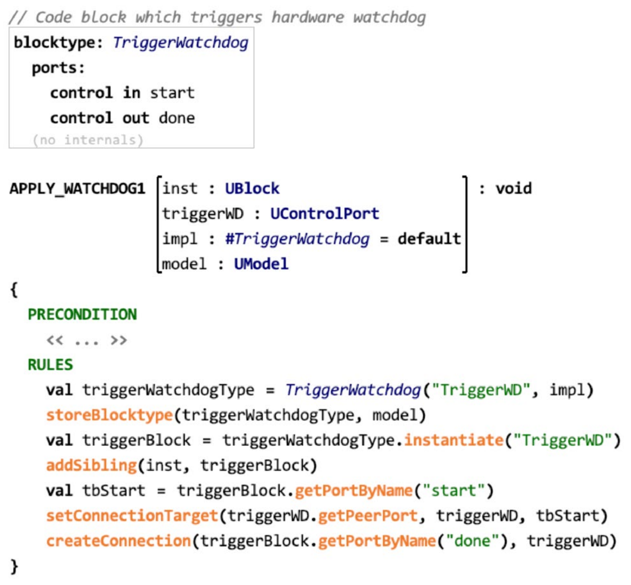
```
void writeActuator(float u) USF trafo: APPLY_WATCHDOG1 [ triggerWD = TriggerWD:IN
                                                         impl = Watchdog_Config_ACC.SetWatchdog ]
{
  // writing to the actuator
  *((float volatile*) (THROTTLE_ACTUATOR_ADDR)) = u;
  ▶ TriggerWD

} writeActuator (function)
```

Fig. 22 Embedded code of the ACC control SW with selected safety mechanism to trigger a hardware watchdog (before transformation)

It just inserted a simple block with some glue code, which retriggers the hardware watchdog timer. Again, the actual code line to do this has been defined in the domain library. If the domain library contains several glue code implementations for different timer hardware, the linked hardware abstraction elements from the USF model (cf. Fig. 11) should be used by the tool to limit the shortlist of implementations to the ones matching the actual timer hardware.

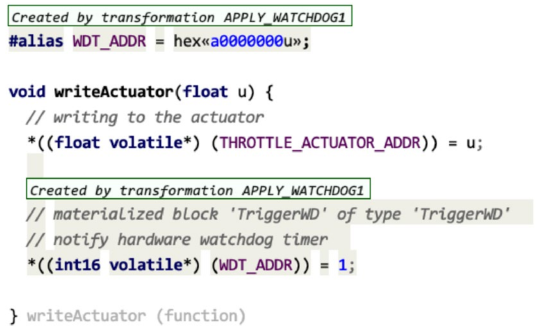
Discussion

With the evaluation, we illustrated how the USF approach can be applied to realize software safety mechanisms. This, of course, requires a significant initial development effort to enable a user-friendly and powerful method to generate safety mechanisms. As previously noted, tool adaptations can be rather complex, but are needed only once per domain. On the other hand, creating new patterns via UTL is quite easy, especially with an appropriate tool



```
// Code block which triggers hardware watchdog
blocktype: TriggerWatchdog
  ports:
    control in start
    control out done
  (no internals)

APPLY_WATCHDOG1 [inst : UBlock                        ] : void
                [triggerWD : UControlPort             ]
                [impl : #TriggerWatchdog = default     ]
                [model : UModel                        ]
{
  PRECONDITION
    << ... >>
  RULES
    val triggerWatchdogType = TriggerWatchdog("TriggerWD", impl)
    storeBlocktype(triggerWatchdogType, model)
    val triggerBlock = triggerWatchdogType.instantiate("TriggerWD")
    addSibling(inst, triggerBlock)
    val tbStart = triggerBlock.getPortByName("start")
    setConnectionTarget(triggerWD.getPeerPort, triggerWD, tbStart)
    createConnection(triggerBlock.getPortByName("done"), triggerWD)
}
```

Fig. 23 UTL example transformation: Hardware watchdog mechanism



```
Created by transformation APPLY_WATCHDOG1
#alias WDT_ADDR = hex«a0000000u»;

void writeActuator(float u) {
  // writing to the actuator
  *((float volatile*) (THROTTLE_ACTUATOR_ADDR)) = u;

  Created by transformation APPLY_WATCHDOG1
  // materialized block 'TriggerWD' of type 'TriggerWD'
  // notify hardware watchdog timer
  *((int16 volatile*) (WDT_ADDR)) = 1;

} writeActuator (function)
```

Fig. 24 Embedded code of ACC control SW with inserted safety mechanism, which triggers a hardware watchdog (after transformation)

support. Nevertheless, to support fully automatic generation of the safe software, newly introduced subcomponents used in the safety patterns (such as comparators) require a provided or generated implementation per domain.

These implementations often represent fairly simple functionality and can be used in multiple patterns, often resulting in modest overhead per domain. In certain cases (e.g., if a pattern requires bare-metal access to a timer without any hardware abstraction layer) several implementations may be required, depending on the used timer. This may lead to a considerable additional effort due to the possible potential of the function, but especially due to the multitude of implementations. Nevertheless, we believe that these development costs will be amortized over a few designs, since hardware, for example, is usually reused.

In order to take full advantage of these simple patterns, an appropriate level of abstraction must be chosen that matches the granularity of the desired safety mechanism. For example, changes at the software architecture level can be easily achieved by applying a simple pattern to a high-level model, while the same changes in the C code would most likely require a complex and non-generic pattern.

For brevity, we have presented the generation of only two patterns in this work. During the development of the USF approach, we analyzed a variety of software safety mechanisms used in industry to create an initial library of safety patterns. These patterns can roughly be divided into application-specific and non-application-specific ones.

The DMR pattern can be classified as an application-specific pattern. Application-specific patterns must be customized and interwoven into the functionality, often deep within the functional software. For this reason, in conventional development, this usually resulted in a manual re-implementation of the safety mechanism rather than a reusable pattern. These steps can now be fully automated with the USF approach.

Patterns that are not application-specific, such as an interrupt controller test, do not need to be customized to the functional design, but must still be called and thus woven into the software at the correct point. The recommendations in safety manuals, often provided by hardware manufacturers for their platforms, are examples of these patterns. Much like conventional development, this requires effort to adapt to a new hardware platform, but the mechanisms can then be shared between projects that use the same hardware. Although the merits of the USF approach for some of these safety mechanisms may be limited to automatic integration of existing implementations or may be useful only in some domains, it can still be very beneficial to express them through patterns. As the example in Figs. 20 and 21 has shown, the use of the USF approach and tools such as the SafetyWeaver allows the user to keep functional and safety software separate, but to combine them at any time following the principle of separation of concerns. Thus, the maintenance of the functional software is simplified. In addition, the description of all mechanisms via the USF pattern enables the creation of a formalized safety specification.

As outlined in Section "Safety Engineering in a Nutshell", the realization of software safety mechanisms is a part of the larger safety engineering process and must always be examined in that larger context. This means, for example, that the safety mechanisms must be developed according to the stringent rules of the safety standards and that suitable patterns are selected by a safety engineer to ensure that the safety goals are met. This includes not only the detection and handling of certain errors, but also other requirements such as the observance of time and memory constraints. Since some safety mechanisms can impose a huge overhead that can conflict, for example, with strict application deadlines, this can be a complicated tradeoff. Automating the realization of safety mechanisms can also assist the safety engineer in this regard through enabling the fast development of various design alternatives that can subsequently be evaluated to check if they fulfill all given requirements.

Related Work

This section on related work is divided into three parts. First, USF and UTL are compared against established concepts. We then compare our approach to other model-driven approaches for generating safety mechanisms, and finally present approaches that focus on the automatic integration of safety mechanisms at the code level.

System Modeling and Model Transformations

There are different benefits of applying domain-specific languages (DSLs) and model-to-model transformations for safety-critical system development. Aside from weaving safety mechanisms into functional code as described here, the language workbench JetBrains MPS has been used for complete generation of safety-critical code and tests from DSL-based models [22]. Another interesting use case is the generation of fault-trees from SysML-like component models using MPS [23].

The mainstream standardized language for modeling systems is SysML [24]. It offers a *profile mechanism* which can be used to specialize its generic metamodel and diagrams in order to support functional safety aspects of systems (e.g., dependability analysis in the aerospace domain [25]). On the other hand, the USF metamodel as a DSL offers the streamlined combination of structural aspects as well as control-/dataflow in the same model. Using a DSL avoids the necessity of using stereotypes, which provides benefits both for manual editing and automated model transformations.

As discussed earlier, automatic safety weaving is implemented as model-to-model transformations supported by the UTL. The general approach is inspired by the aspect-oriented programming (AOP [13]) methodology. However, USF patterns might be applied to different target domains (not only to source code of a single programming language). Therefore, USF safety weaving is not equivalent to AOP.

For the definition of transformations on generic metamodels, a variety of languages and corresponding implementations has been developed. QVT [26] is a standardized transformation/query language operating on models, which conform to MOF 2.0. ATL [14] is a QVT-like language for EMF models. Viatra2 [27] is also a query/transformation language operating on EMF models, but with a high-performance incremental implementation. Xtend [17] is a general-purpose programming language with special focus on model-to-text generation and useful language concepts for model-to-model transformations (e.g., create methods, builder syntax and higher-order functions). As shown in Section "Requirements for the USF Transformation Language", many of the requirements of UTL are specific to the task of safety weaving and the needs of safety engineers. Therefore, we preferred designing a DSL tailored for this task and the corresponding user group.

Model-Driven Safety Mechanism Generation

The main advantage of model-driven approaches is that even complex safety mechanisms can be incorporated into the design at an early stage of development and, moreover, are often independent of the target platform. Since the model-based designs are usually easier to understand than, for example, the plain source code, this also potentially eases the validation effort for the safety engineer. There are several model-driven approaches that automatically integrate software safety mechanisms via model transformations but are limited to a specific type of safety mechanism and a specific modeling language. Trindade et al. [5], for example, introduce a technique to generate boundary checks for AUTOSAR software components from semi-formal requirements. In [4] Hu et al. propose an approach to applying N-version programming in the Cyber-Physical Action Language and Ding et al. [3] presented a flow to integrate different computational redundancy mechanisms, e.g., DMR, into Simulink models.

A more general approach is presented by Huning et al. [28]. The authors present a workflow for applying safety patterns to UML models, which are then used to generate code. Adding new safety patterns is a multi-step process. This includes the creation of new stereotypes, model-to-model and model-to-text transformations for each pattern. While this approach is limited to UML models, there are similarities between this concept and our approach. However, there does not appear to be any additional tool support for creating new patterns beyond what is provided by standard UML and model transformation tools. Making it significantly more laborious to add new patterns compared to our approach.

Code Transformation Methods

The model-based USF flow can be applied at source code level using the *SafetyWeaver* tool. The transformations integrate (weave) safety-related source code snippets at the defined places into the functional source code based on the USF transformation language. These code modifications are source-to-source (S2S) code transformations. They could also be used with other C/C++ frameworks that allow S2S transformations such as LLVM [29] or the Rose Compiler [30].

Code transformations for safety can also be used to implement so-called SW-implemented HW fault tolerance (SIHFT) methods. These methods add either dual redundancy to the software program to detect transient hardware errors that lead to data corruption in the processor, or they add signatures to code basic block to detect corruptions in the control flow of the program. Different SIHFT variants exist to protect computation [6, 7, 31] as well as conditions [8] and control flow [9]. A major problem for implementing SIHFT methods is that the compiler can detect and remove redundant computations. Hence, these techniques are best added at the assembly code level during the backend code generation of the compiler. These methods can be integrated in the presented model-based safety flow in a straightforward way. USF transformations add additional markers to source code sections such as functions to indicate to the compiler that a specific SIHFT method should be applied to harden this code section against random hardware errors.

Conclusions

In this work, we demonstrated a model-driven approach to automatically adapt, generate, and integrate domain-specific software safety mechanisms using the Universal Safety Format. Safety mechanisms are generalized by patterns described via the domain-agnostic transformation language UTL, which operates on USF models. Once created, these safety patterns can be reused in various designs and at different design stages. We have demonstrated how USF support can be integrated into domain-specific tools such as the SafetyModeler and the SafetyWeaver, which can then apply USF safety patterns in a domain context to realize the software safety mechanisms. From our evaluations, we have shown how this can be implemented for very different domain contexts, such as Simulink models or C code, using the same patterns. Additional information and open source implementations can be found on the USF website [12].

In future work, we are investigating how this approach can be extended to the generation of security mechanisms as well as how the approach can be adapted to hardware designs.

Funding Open Access funding enabled and organized by Projekt DEAL. This work was supported by the German Ministry of Science and Education (BMBF) in the project SAFE4I under Grant No. 01IS70320.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. IEC 61508. Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems (E/E/PE, or E/E/PES. Geneva, CH: The International Electrotechnical Commission, 2010.
2. ISO 26262. Road vehicles – Functional safety. Geneva, CH: International Organization for Standardization; 2018.
3. Ding K, Morozov A, Janschek K. MORE: MOdel-based REdundancy for Simulink. In: Gallina B, Skavhaug A, Bitsch F, editors. Computer safety, reliability, and security. Cham: Springer; 2018. p. 250–64.
4. Hu T, Cibrario Bertolotti I, Navet N, Havet L. Automated fault tolerance augmentation in model-driven engineering for CPS. Comput Standards Interface. 2020;70: 103424. https://doi.org/10.1016/j.csi.2020.103424.
5. Trindade RFB, Bulwahn L, Ainhauser C. Automatically generated safety mechanisms from semi-formal software safety requirements. In: Bondavalli A, Di Giandomenico F, editors. Computer safety, reliability, and security. Cham: Springer; 2014. p. 278–93.
6. Didehban M, Shrivastava A. NZDC: A Compiler Technique for near Zero Silent Data Corruption. In: Proceedings of the 53rd Annual Design Automation Conference. DAC '16. 2016;New York, NY, USA: Association for Computing Machinery; p. 1–6.
7. Reis GA, Chang J, Vachharajani N, Rangan R, August DI. SWIFT: Software Implemented Fault Tolerance. In: International Symposium on Code Generation and Optimization. IEEE. 2005;p. 243–254.
8. Didehban M, Shrivastava A, Lokam SRD. NEMESIS: A software approach for computing in presence of soft errors. In: 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2017;p. 297–304.
9. Vankeirsbilck J, Penneman N, Hallez H, Random BJ. Additive signature monitoring for control flow error detection. IEEE Trans Reliab. 2017;66(4):1178–92. https://doi.org/10.1109/TR.2017.2754548.
10. Armoush A. Design patterns for safety-critical embedded systems [Ph.D. thesis]. RWTH Aachen University; 2010.
11. Haxel F, Viehl A, Benkel M, Beyreuther B, Birken K, Schmedes R, et al. Universal Safety Format: Automated Safety Software Generation. In: Pires LF, Hammoudi S, Seidewitz E, editors. Proceedings of the 10th International Conference on Model-Driven Engineering and Software Development, MODELSWARD

2022, Online Streaming, February 6–8, 2022. SCITEPRESS. p. 155–166.
12. USF.: Universal Safety Format - Website. Last checked on May 20, 2022. Available from: https://www.universalsafetyformat.org/.
13. Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes CV, Loingtier J, et al. Aspect-Oriented Programming. In: Aksit M, Matsuoka S, editors. ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9–13, 1997, Proceedings. vol. 1241 of Lecture Notes in Computer Science. Springer; 1997. p. 220–242.
14. Jouault F, Allilaire F, Bézivin J, Kurtev I. ATL: A model transformation tool. Sci Comput Program 2008;72(1):31–39. Special Issue on Second issue of experimental software and toolkits (EST). https://doi.org/10.1016/j.scico.2007.08.002.
15. MPS.: Meta Programming System (MPS) by JetBrains. Last checked on May 20, 2022. Available from: https://www.jetbrains.com/mps/.
16. Knüchel C, Rudorfer M, Voget S, Eberle S, Sezestre R, Loyer A. Artop an ecosystem approach for collaborative AUTOSAR tool development. In: ERTS2 2010, Embedded Real Time Software & Systems.
17. Xtend.: Xtend programming language homepage. Last checked on May 20, 2022. Available from: http://www.eclipse.org/xtend.
18. Voelter M. Generic Tools, Specific Languages [Ph.D. thesis]. Delft University of Technology, 2014.
19. Voelter M, Ratiu D, Kolb B, Schaetz B. mbeddr: Instantiating a Language Workbench in the Embedded Software Domain. Autom Softw Eng. 2013;20(3):339–90. https://doi.org/10.1007/s10515-013-0120-4.
20. Voelter M, Birken K, Lisson S, Rimer A. Shadow Models: Incremental Transformations for MPS. In: Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering. SLE 2019. New York, NY, USA: Association for Computing Machinery; p. 61–65.
21. Sanchez B, Zolotas A, Hoyos Rodriguez H, Kolovos D, Paige R. On-the-Fly Translation and Execution of OCL-Like Queries on Simulink Models. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS); 2019;p. 205–215.
22. Voelter M, Kolb B, Birken K, Tomassetti F, Alff P, Wiart L, et al. Using language workbenches and domain-specific languages for safety-critical software development. Softw Syst Model. 2019;18(4):2507–30. https://doi.org/10.1007/s10270-018-0679-0.
23. Munk P, Nordmann A. Model-based safety assessment with SysML and component fault trees: application and lessons learned. Softw Syst Model. 2020;19(4):889–910. https://doi.org/10.1007/s10270-020-00782-w.
24. Mann C. A Practical Guide to SysML: The Systems Modeling Language. Kybernetes, 2009;38.
25. Steurer M, Morozov A, Janschek K, Neitzke KP. SysML-based Profile for Dependable UAV Design. IFAC-PapersOnLine. 51(24):1067–1074. 10th IFAC Symposium on Fault Detection, Supervision and Safety for Technical Processes SAFEPROCESS, 2018. https://doi.org/10.1016/j.ifacol.2018.09.722.
26. Bast W, Murphree M, Michael L, Duddy K, Belaunde M, Griffin C, et al. MOF QVT final adopted specification: meta object facility (MOF) 2.0 query/view/transformation specification. Object Management Group, 2005.
27. Bergmann G, Ujhelyi Z, Ráth I, Varró D. A Graph Query Language for EMF models. In: Cabot J, Visser E, editors. Theory and Practice of Model Transformations, Fourth International Conference, ICMT 2011, Zurich, Switzerland, June 27-28, 2011. Proceedings. vol. 6707 of Lecture Notes in Computer Science. Springer. Springer; p. 167–182.
28. Huning L, Iyenghar P, Pulvermüller E. A Workflow for Automatically Generating Application-level Safety Mechanisms from UML Stereotype Model Representations. In: Ali R, Kaindl H, Maciaszek LA, editors. Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2020, Prague, Czech Republic, May 5–6, 2020. SCITEPRESS; 2020. p. 216–228.
29. Lattner C, Adve V. LLVM: a compilation framework for lifelong program analysis amp; transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004.; p. 75–86.
30. Quinlan D, Liao C. The ROSE source-to-source compiler infrastructure. In: Cetus users and compiler infrastructure workshop, in conjunction with PACT. vol. 2011. Citeseer; p. 1.
31. Sharif U, Mueller-Gritschneder D, Schlichtmann U. REPAIR: Control Flow Protection Based on Register Pairing Updates for SW-Implemented HW Fault Tolerance. ACM Trans Embed Comput Syst. 2021;20(5s). https://doi.org/10.1145/3477001.