

FASTEN: An Extensible Platform to Experiment with Rigorous Modeling of Safety-Critical Systems

Daniel Ratiu, Arne Nordmann, Peter Munk, Carmen Carlan, Markus Voelter

Abstract The increasing complexity of safety critical systems and the shorter time-to-market requires a high degree of automation during all development phases from requirements specification to design, implementation, verification and safety assurance. To make this feasible, we need to describe different system aspects using appropriate models that are semantically rich and, whenever possible, formally defined such that they are verifiable by automated methods. At the same time they must be easy to understand by practitioners and allow them to capture the domain concepts with minimal encoding bias. In this chapter, we describe FASTEN, an open source research environment for model-based specification and design of safety critical systems using domain specific languages. FASTEN enables the experimentation with modeling abstractions at different levels of rigor and their integration in today's development processes. We present an overview of the currently available domain specific languages (DSLs) used to formally specify requirements, system designs and assurance arguments. These DSLs have been developed and used in technology transfer projects by researchers from different organizations – Siemens, Bosch, fortiss and itemis. Last but not least, we discuss lessons learned from implementing the languages and interacting with practitioners and discuss the language engineering features of MPS that enabled our approach and its open challenges.

Daniel Ratiu
VW Car.Software.Org (previously with Siemens), e-mail: ratiud@googlemail.com

Arne Nordmann
Robert Bosch GmbH, Stuttgart, e-mail: arne.nordmann@de.bosch.com

Peter Munk
Robert Bosch GmbH, Stuttgart, e-mail: peter.munk@de.bosch.com

Carmen Carlan
fortiss, Munich, e-mail: carmen.carlan@fortiss.org

Markus Voelter
independent/itemis, Stuttgart, e-mail: voelter@gmail.com

1 Introduction

It is common knowledge that software-intensive systems in general are becoming larger, more complex and more relevant to crucial tasks in our society. Due to the high impact of a malfunctioning, we must have a high degree of confidence that the systems cannot harm people or expensive equipment [14].

Testing is Limited. Testing is a well-known approach to building trust. Systems are “tried out”, unit tests and integration tests are written and automatically executed for the software parts, hardware-in-the-loop tests verify aspects of the hardware and red teams try to attack the system to uncover vectors for malicious attacks. However, usually testing can only show the presence of bugs, and not prove their absence. Phrased differently, testing suffers from the coverage problem, which means that you can only be sure that your system is “correct” if you test it completely. “Completely” is a high bar that is often not reachable in practice for complex systems.

Formal Methods. Formal methods can be an important ingredient in an engineer’s toolset to build trust in critical systems. Depending on the particular formalism, formal methods can either help with systematically improving the coverage of tests or can even prove the *absence* of certain classes of errors such as runtime errors (e.g. overflows) or conformance of a client’s implementation with a API. Some formal verification tools (e.g. cbmc [9]) work directly on source code, however, most require a model expressed in a particular language on which to operate. While this can be seen as a disadvantage (if you are a code-centric developer), it has the important advantage that models cannot just represent software – they can also represent aspects of the system implemented in hardware, or even aspects of the environment. Models – for example of interfaces, protocols or state-based behavior – can also be defined in earlier stages of development where hardware or source code are still elusive. This way, engineers can experiment with various design alternatives *early* in the development, building trust in their work *early*, and avoid expensive rework during later stages of development.

Bringing Formal Methods Closer to Practitioners. However, formal methods are hard to use by practitioners for several reasons. First, some of the formalisms are conceptually hard to understand [21]; they often encode non-trivial mathematical ideas that are not familiar to engineers [31]. The input languages of verification tools contain low-level abstractions that are targeted towards verification, which forces engineers to bridge a large abstraction gap when they encode system-level concepts. Second, these formalisms are by necessity general – they are not specific to the engineer’s domain, which makes the transformation of the engineering model to the tool’s input and the lifting of the results even harder. Third, there is often no robust tool support (IDEs) for verification engines; and common software IDE services such as auto-completion, refactorings or debugging the models is non-existent. Fourth, using real-world verification requires the use of *multiple* formalisms for the definitions of state-transitions or constraints, requiring multiple encodings of the engineering model and/or fusion of the results. Fifth, the interpretation of the results of the verification tools, such as understanding the witnesses for verification failures in

terms of the engineering model, is often not trivial either. In safety-critical contexts, formal verification results may be used as evidence supporting assurance arguments that demonstrate that the system meets critical goals. Finally, not everything that is needed to make an argumentation for the system's safety can be formalized [34, 13]. In these cases, unstructured or semi-structured artifacts (such as the original textual requirements, SysML diagrams) must be integrated with formalized models, both conceptually and technically.

Our Vision. We envision an integrated modeling and verification platform, that deeply integrates models for requirements, design, verification and assurance at increasing levels of formality as illustrated in Figure 1. Our platform has the following characteristics: (1) The user interacts with a limited number of models whose structure and notation is meaningful to the user's engineering domain. Informal parts of the system such as textual requirements, or safety arguments can be incrementally formalized and combined with other formal specifications. (2) The languages used to define these models allow the user to express properties that they want to verify; again, these properties are expressed with a language that is close to the user's domain. (3) These models, together with the properties they must satisfy, are then automatically translated into one or more verification formalisms, and (4) existing verification engines are executed to verify the properties. (5) The low-level verification results are lifted back to the level of the engineering model; potentially, the results from multiple verification tools are semantically integrated. (6) Using references and other model-level mechanisms, the formal models can be connected to informal or semi-formal content, (7) integrating system and safety engineering models in a semantically rich assurance case to ensure consistency between design and safety models. Last but not least, the tool should be built as an open platform to make it extensible with new formalisms or user-facing languages, and its user experience should be on par with modern IDEs in terms of editor features, type checking and error reporting.

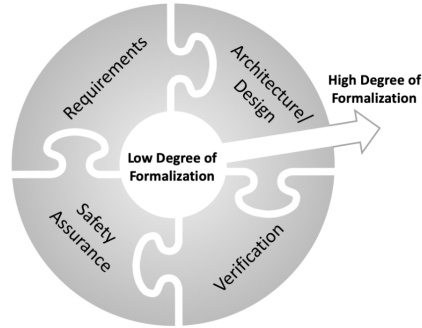


Fig. 1 FASTEN Vision: Deeply integrated models for requirements, design, verification and safety assurance. The models support transition from informal to formal system representation.

How to get there. We rely on language workbenches [12], tools that support the efficient implementation of languages, type systems, model transformations, and IDEs. We use a layered approach that delivers early benefits even while only a part

of our overall vision is implemented. As foundational language workbench we chose JetBrains MPS¹ due to its powerful support for language engineering (Section 7.2).

We start with the implementation of several input languages of verification tools in our language workbench – e.g. we implement the language SMV, the input language of the NuSMV [7] model checker; or Promela, the input language of Spin [15] model checker. This step does not give us improvements in terms of semantic abstraction, but it results in a robust IDE for writing models in the notation of the formalism that has the usual modern front-end features such as syntax coloring, code completion, type checking and reporting of the verification results. Based on JetBrains MPS’ support for modular language extension, we incrementally add discrete extensions to these low-level input languages to make idiomatic use simpler. These extensions are still generic, but useful for less mathematically-minded users.

Next, we implement an integrating language based on the component-instance-connector paradigm; such languages are well known to many engineers, provide good support for hierarchical breakdown of systems and are reasonably generic (as evidenced by SysML [27]). Furthermore, we develop extensions to the component language that allow the user to annotate properties relevant for verification. A chain of model-to-model transformations convert this model, including the properties to verify, into the input language of one of the integrated verification tools. After the verification is run, we provide lifting of verification results back to the users such that they can easily understand what went wrong and perform fixes. This integrated language for modeling and verification is the first major goal of our vision.

To enable a transition from textual requirements to formal models, we have developed a set of DSLs for specifying requirements by using increasingly semantically rich models. The requirements models range from plain natural language text to requirements templates (aka. boilerplates) or formal models written e.g. using temporal-logics. The richer the models the more rigorous verification is possible. The results of verification can be further used in safety assurance arguments that we integrate via another set of DSLs specialized for safety engineering.

Are we there yet? FASTEN² is an open-source³ platform that enables experimentation with modeling abstractions amenable for verification on the way to our vision. It supports exploration of the idea, in a bottom-up manner by combining informal and increasingly formal models, and verifies the degree to which it is realistic. FASTEN is built on JetBrains MPS, which has been used successfully in a safety-critical context [35]. While FASTEN is not a production-ready tool, it has been used to verify realistic systems in industrial settings. To validate the extensibility with regards to verification formalisms, we present various extensions shipped with FASTEN itself. In addition to these open-source extensions, we also discuss closed-source extension, developed independently at Bosch. To validate the extensibility for a particular domain, we demonstrate a more extensive case study developed at the Corporate Research department of Bosch.

¹ <https://www.jetbrains.com/mps/>

² <https://sites.google.com/site/fastenroot/home>

³ <https://github.com/mbeddr/mbeddr.formal>

Contributions. The main contribution of this chapter is FASTEN, an open-source platform based on JetBrains MPS, for safety critical systems development. FASTEN allows experimentation with adequate and domain specific modeling abstractions to capture different aspects of safety critical systems from requirements, design, verification to safety assurance. The parts focused on safety assurance is referred as FASTEN.Safe. For each of these aspects, we provide DSLs that enable the transition from informal to formal descriptions. We present a novel architecture for building model driven engineering tooling around modular and extensible stacks of DSLs that leverage on the language engineering capabilities of the JetBrains MPS language workbench. FASTEN has been built over the last three years by industrial researchers from three companies (Siemens, Bosch, itemis) and a research institute (fortiss). Last but not least, this chapter presents our experiences and lessons learnt with developing and using FASTEN in research transfer projects inside our organizations.

Structure. In Section 2, we present the FASTEN-platform, which is the infrastructure that serves as a basis for our DSLs. Section 3 presents DSLs for modeling requirements at different levels of rigor from natural language text to boilerplates or formal models. Section 4 presents DSLs for design and specification developed on top of SMV, the input language of the NuSMV model checking engine. Section 5 presents DSLs for modeling safety assurance arguments and linking them with other system models to enable automated consistency checks. Section 6 presents extensions and applications developed by Bosch. We conclude this chapter by presenting our lessons learned in Section 7 and conclude it in Section 8.

2 The FASTEN Platform

To enable an efficient implementation of our vision, we have modularized recurring functionality in a set of DSLs and libraries that make up the *FASTEN Platform*. Figure 2 shows an overview of the FASTEN architecture. FASTEN integrates external analysis engines (see bottom of the figure) as black boxes – NuSMV [7], Spin [15] and Prism [22] are integrated as external binaries, the Z3 [10] is integrated via its Java API.

Foundational Libraries. All languages and functionalities are developed on top of JetBrains MPS plus languages and libraries provided by the *MPS-extensions*⁴ and *mbeddr-platform*⁵ projects. We use these language libraries for diagrammatic, tree and tabular notations, for the improving editor usability (via grammar-cells [38]) and for generated code review. From an implementation perspective this infrastructure proved to be crucial to our approach.

Our Infrastructure: the FASTEN Platform. In order to facilitate the development of DSLs and the integration of formal analysis tools, FASTEN comes with a set of basic languages and functionalities that can be grouped as follows:

⁴ <https://github.com/JetBrains/MPS-extensions>

⁵ <http://mbeddr.com/platform.html>

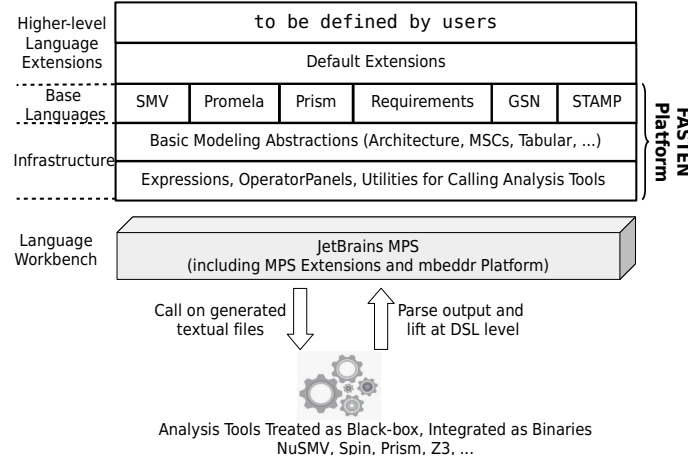


Fig. 2 FASTEN architecture features layers of DSLs (top) built on the JetBrains MPS language workbench (middle). External analyses tools (bottom) are integrated at binary level. At its base, FASTEN provides itself a platform of reusable DSLs (FASTEN Platform) that contains an implementation of an expressions language, utilities for generic modeling abstractions (e.g. for modeling architecture, MSCs) and, based on these, the implementation of standard modeling languages (e.g. GSN, STAMP) and of base-languages of analysis engines (e.g. SMV, Promela). On top of the platform, FASTEN provides a set of extensions as presented in the following sections.

1. *Infrastructure*: commonly used DSLs such as an expressions language; support for calling external tools, displaying analysis results in the IDE,
2. *Basic modeling*: base DSLs for the definition of modeling languages such as architecture, message-sequence-charts or tabular specifications,
3. *Input languages of existing analyses tools*: the implementation of input languages (e. g., SMV, Promela, SMTlib, Prism) of the integrated analysis tools.
4. *Standard languages*: the GSN [18] modeling language for creating assurance cases, the STAMP [23] modeling language for performing hazard analysis and a base language for the specification of textual requirements.

On top of the *FASTEN-Platform*, we have created DSL extensions to enable more comfortable modeling and verification. In the following sections we present selected examples of higher-level abstractions and functionalities that have been built so far.

3 Modeling of Requirements

As presented in the introduction, one of the essential ingredients of our vision is to enable a seamless transition from informal to formal specifications. This is especially crucial for requirements, because in industry, they are usually written as mostly plain natural language. To enable this, we defined a set of DSLs that allow users to write requirements in an increasingly rigorous manner, starting from plain natural language up to formal models. In Figure 3 we illustrate our current stack of DSLs for requirements modeling.

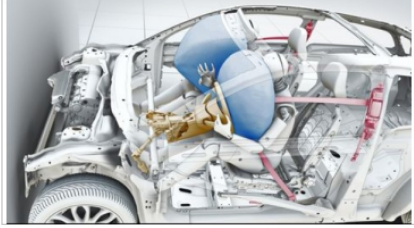
Formal Specification	Temporal Logics	SMV Models
Semi-formal Specification	Tracing to Models	Sentence Templates
Informal Specification	Requirements Base Language (Natural Language)	

Fig. 3 Stackable DSLs for modeling requirements and enabling transition from informal to (semi-)formal specifications.

Textual Requirements (Natural Language). At its lowest level, requirements are specified using plain natural language text and pictures. Traces can be created between requirements or other models. In Figure 4 we present an example of textual specification of an airbag function that references other requirements.

Req 010.01 : Airbag functionality overview
created by: john **kind:** functional

An airbag shall protect the passengers of a car in case of impact. The figure below illustrates intuitively the deployment of airbags. Requirement @req(010.02) contains the specification of the interface between the car sensors and the airbag system.



press enter to add sub-requirement

Fig. 4 Example of textual specification of requirements. Besides the text, a requirement might contain pictures or references to other requirements.

Semi-formal Specification (Sentence Templates). A lot of work has been done on structuring requirements by expressing them in controlled natural language (CNL) [33]. There, requirements are expressed as a set of boilerplates (natural language sentence templates) which are instantiated for a given system. FASTEN provides boilerplates for properties over traces (aka. temporal patterns). In Figure 5 we present examples of templates mixed with natural language text for specifying an airbag.

Req 020.01 : Airbag high-level functions
created by: john **kind:** functional

Airbag requirements are specified below using sentence boilerplates for temporal patterns

```
temporal logics specification {
  Globally, it is always the case that %airbag is ready to deploy in case of impact% holds.
  After %impact was detected%, it is always the case that if %impact severity is high% holds
    , then %the airbag will be deployed% eventually holds.
}
```

press enter to add sub-requirement

Fig. 5 Example of specification of requirements using boilerplates mixed with unstructured text.

Formal Specification. This allows the specification of requirements using formal models with a mathematically defined meaning. FASTEN allows users to write requirements using temporal logics by providing DSLs for commonly recurring patterns [11], as illustrated in Figure 6. Furthermore, FASTEN allows referencing formal models from within requirements specifications, as exemplified in Figure 7, which defines the activation logic of the airbag. Using the editor capabilities of MPS, fragments of formal models can be displayed (projected) where the textual requirements are defined. By displaying the formal model in the proximity of the textual requirement from where it originates we ease the validation of the model.

```

Req 030.01 : Airbag protects the passengers in accidents
created by: dan kind: interface


---


The input port collision_detected is TRUE when sensors detect a collision.
The airbag should explode whenever a collision is detected - as formalized in the following.
¶
in scope airbag the following properties hold {
  Globally, collision_detected eventually holds.
  After collision_detected, it is always the case that airbag_explode_command holds.
}
press enter to add sub-requirement
of
Req 030.02 : A deactivated airbag shall not explode
created by: dan kind: interface


---


The input port deactivate_airbag is TRUE when the airbag was deactivated by the driver.
After an airbag is deactivated it shall not explode - as formalized in the following.
¶
in scope airbag the following properties hold {
  Globally, deactivate_airbag eventually holds.
  -- this requirement is in contradicton with the one from above;
  After deactivate_airbag, it is always the case that !airbag_explode_command holds.
}

```

Fig. 6 Example of specification of requirements using temporal logics patterns defined by [11]. The patterns formalize the textual requirements and thereby enable automated consistency checks.

```

Req 050.01 : Airbag basic function
created by: dan kind: interface


---


If not in a deactive state, the airbag shall explode whenever a collision
is detected - as formalized in the following.
¶
>> Traced-node spec: airbag
MODULE airbag(deactivate_airbag, collision_detected) {
  DEFINE {
    output airbag_explode_command := explode_state;
  }
  VAR {
    deactivated_state : boolean;
    explode_state : boolean;
  }
  ASSIGN {
    init(deactivated_state) := FALSE;
    next(deactivated_state) := deactivated_state | deactivate_airbag;

    init(explode_state) := FALSE;
    next(explode_state) := explode_state | (!deactivated_state & collision_detected);
  }
}

```

Fig. 7 Example of tracing formal models from requirements. In this example we present the activation logic of the airbag specified as an SMV module. The traced nodes can be projected directly in the requirements document.

4 Formalizing System-Level Designs with SMV-based DSLs

In this section, we present how we formalize system-level designs by using DSLs built on top of SMV, the input language of the NuSMV [7] model checker. Compared to the other verification engines integrated in FASTEN (e.g. Spin and Prism), the integration of NuSMV is the most advanced both in terms of tooling as well as in terms of higher-level DSLs. Figure 8 shows an overview of the stack of DSLs developed on top of SMV. At the bottom of our stack is an implementation of SMV itself. Thus, in its most basic usage scenario, FASTEN could be seen as a front-end for developing SMV models. On top of SMV we have been "growing" layers of DSLs that are at higher abstraction levels to make systems' modeling easier. A detailed discussion of the SMV extensions available in FASTEN is presented in [29].

Higher-level Modeling	Contracts	Contract-based Design		Verification Cases	Generalized Tests
"Simple" Extensions	Architecture	State-Machines	Tabular Spec	Scenarios	Tests
Convenience	SMV Convenience Extensions (e.g. adding types to module parameters, typedef)				
Base	SMV Base Language				

Fig. 8 On top the SMV language, there are layers of DSLs to facilitate the specification of architectures (e.g. contracts), behavior (e.g. state-machines, tables), or of properties (e.g. scenarios, generalized tests, temporal logic patterns)

Basic Extensions. To conveniently work with SMV models we have developed extensions for 1) adding types to module parameters; 2) defining structured types, and 3) declaring user-defined types (aka. typedefs). In Figure 9 we illustrate examples of these extensions. The annotated types are used for IDE-level checks and automation but are removed when textual SMV code is generated (SMV does not support the

```

typedef 0..100 as Time ;
typedef 0..100 as Speed ;

struct Position {
  x : 0..100;
  y : 0..100;
}

MODULE emergency_braking(crt_speed_x : Speed, car_pos : Position) {
  VAR {
    old_car_pos : Position;
  }
  ASSIGN {
    next(car_pos.x) := car_pos.x + crt_speed_x;
    next(car_pos.y) := car_pos.y;
    next(crt_speed_x) := car_pos;
  }
}

```

Error: type Position is not a subtype of Speed

Fig. 9 Example of SMV-basic extensions (marked in figure with red). We have added support for user-defined types, for structured types or explicitly specifying types of module parameters.

definition of types for parameters of modules). Despite the fact that both **Time** and **Speed** are reduced to SMV as interval types, variables with **speed** type cannot be assigned to those with **time** type.

(Generalized) Tests. The most commonly used verification method by practitioners is testing. Thus, we have implemented DSLs extensions for writing tests for modules as illustrated in Figure 10-left. In this example, the system under test is a **counter** module that has two inputs (**stop_cmd** and **step**) and two outputs (**out_valid** and **out_value**); if the input **stop_cmd** is true, the counter will stop counting and set its output **out_valid** to **FALSE**.

On the lower part we illustrate the translation of test cases to SMV. We provide a counter (**__crtStep**) representing the current step in the execution of the test. Based on the current step we select the inputs of the module (as illustrated by the **stop_cmd** and **step** defines) and the expected outputs.

A test case represents a single run of a system with specific inputs – the coverage of the input space by simple test-cases is very limited. To increase coverage efficiently, we have extended the testing language with *generalized* tests (Figure 10-right) that contain '*' (stars) as inputs. These mean that the value of the input in a certain step is taken from an interval. This way each generalized test describes a family of test cases that cover an input space equivalent to thousands or even millions of tests. For the cases in which the value of an output is irrelevant, users can write '#' (meaning "Don't care" or "Discard"). We implement the 'asterisk' using special SMV variables

test case: test_1 for module: counter {					
#	Inputs		Outputs		
	stop_cmd	step	out_valid	out_value	
1	FALSE	1	TRUE	0	
2	FALSE	2	TRUE	1	
3	TRUE	3	TRUE	3	
4	FALSE	1	FALSE	3	
5	FALSE	5	FALSE	3	
}					
test case: test_2 for module: counter {					
#	Inputs		Outputs		
	stop_cmd	step	out_valid	out_value	
1	FALSE	1	TRUE	0	
2	FALSE	2	TRUE	1	
3	TRUE	*	TRUE	#	
4	*	*	FALSE	#	
5	*	*	FALSE	#	
}					

<pre> MODULE main VAR __crtStep : 0..100; cnt : counter(stop_cmd,step); DEFINE stop_cmd := ((__crtStep = 0) ? FALSE: (__crtStep = 1) ? FALSE: ...; step := ((__crtStep = 0) ? 1: (__crtStep = 1) ? 2: ... ASSIGN init(__crtStep) := 0; next(__crtStep) := (((__crtStep + 1) < 5) ? (__crtStep + 1):(5); LTLSPEC ((cnt.out_valid=TRUE & cnt.out_value=0)) & (X (cnt.out_valid = TRUE & cnt.out_value = 1)) & (X X (cnt.out_valid = TRUE & cnt.out_value = 3)) ... </pre>	<pre> MODULE main VAR __crtStep : 0..100; cnt : counter(stop_cmd, step); stop_cmd_nondet : boolean; step_nondet : 0..10; DEFINE stop_cmd := ...((__crtStep=3) ? stop_cmd_nondet: ...; step := ...((__crtStep = 2) ? step_nondet: ... ASSIGN init(__crtStep) := 0; next(__crtStep) := (((__crtStep + 1) < 5) ? (__crtStep + 1):(5)); LTLSPEC ... (X X (cnt.out_valid=TRUE & cnt.out_value=3)) & (X X X (cnt.out_valid = FALSE))... </pre>
---	--

Fig. 10 FASTEN features classical tests (left) or generalized tests which have ranges for certain inputs (right). On the lower part of the figure is exemplified the translation of these DSLs to SMV.

generated for the corresponding inputs that are 'unconstrained', meaning that they can take all values allowed by their type. An example is shown in Figure 10-right.

Allowed/Disallowed Scenarios. When the system under verification contains non-determinism, using tests to check its behavior does not make sense since due to the non-determinism for certain input values various output values are possible. For these situations, FASTEN supports the definition of allowed/disallowed scenarios. Figure 11 illustrates a scenario definition for a traffic lights controller – when the signal `ped_request` is true, the `ped_signal` will eventually allow walking across the street. Note that the translation of scenarios to SMV is similar to the translation of tests, but with a different property specification: because a scenario is a *possible* trace through the system (which due to non-determinism might or might not happen), in order to fully check the feasibility of a scenario we negate the entire formula in SMV. If the verification with NuSMV is successful, then the property (specifying the allowed scenario) fails since there is no trace with the given values.

allowed scenario: pedestrian allow		
#	Inputs	Outputs
	ped request	ped signal
1	FALSE	DontWalk
2	TRUE	DontWalk
3	FALSE	DontWalk
4	FALSE	DontWalk
5	FALSE	DontWalk
6	FALSE	DontWalk
7	FALSE	DontWalk
8	FALSE	Walk

MODULE harness
VAR
 __crtStep : 0..100;
 tlc : traffic_lights_controller(ped_request);

DEFINE
 ped_request := ((__crtStep = 0) ? FALSE:
 (__crtStep = 1) ? FALSE: ...;

ASSIGN
 init(__crtStep) := 0;
 next(__crtStep) := (((__crtStep + 1) < 9) ?
 (__crtStep + 1):(9));

LTLSPEC !(tlc.ped_signal = DontWalk &
 (X (tlc.ped_signal = DontWalk)) & ...

Fig. 11 Examples of high-level model that describes allowed scenarios and the translation of scenarios to SMV. The actual parameters values of the module under test, `tlc` are set based on the value of the `__crtStep`; the LTLSPEC checks on the output `ped_signal` of the `tlc` module.

Architecture, Contracts. The basic modularization mechanism of SMV are modules. A SMV module has input parameters, a set of (typed) internal state variables and a set of definitions (DEFINES) which represent aliases for expressions. A module can be used as the type of state variables (Figure 12-top presents three variables with type `controller` and one with type `voter_2oo3`). SMV does not specify outputs of modules, all variables of modules and DEFINES can be accessed globally, which breaks encapsulation. In order to describe the architecture, we extended the SMV language with special kinds of define that are outputs of modules. We also specialized the VAR section with a new concept WIRING. All variables inside a WIRING can take only module types. A WIRING thus defines a composite component that can be projected as a diagram. In Figure 12 we exemplify the definition of a three-channel-architecture for a braking controller both in textual as well as in diagrammatic notation.

```

MODULE system(speed1, dist1, speed2, dist2, speed3, dist3) {
  DEFINE {
    output break_cmd := v.break_cmd;
  }
  WIRING {
    c1 : controller(speed1, dist1);
    c2 : controller(speed2, dist2);
    c3 : controller(speed3, dist3);
    v : voter_2oo3(c1.break_cmd, c2.break_cmd, c3.break_cmd);
  }
}

```

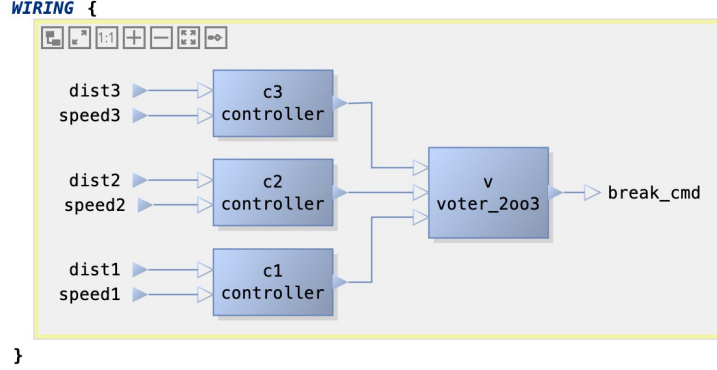


Fig. 12 Examples of a WIRING section for modeling the architecture (textual notation on top and diagrammatic notation on the bottom). The example presents a three channels architecture for an emergency braking controller – the voting component issues a braking command only when at least two out of three controllers issue this command.

Once we have defined the syntactic interface of components (inputs and outputs) the next step is to add semantics via contracts written using temporal logics or a higher-level extension (e.g. temporal logics patterns). In Figure 13-top we exemplify how FASTEN users can attach contracts to modules – in the example the contracts refer to the conditions for the explosion of an airbag. The contracts are subsequently translated to SMV as LTL formula (LTLSECS) – Figure 13-bottom.

```

pre(1) collision_pre : G (collision -> X collision);
post(1) exploded_only_by_collision : G (explode_cmd -> O collision);
post(2) explode_when_collision : G (collision -> F explode_cmd);
MODULE airbag_controller(collision) {
  DEFINE {
    output explode_cmd := explode;
  }
  VAR { ... }
}
-----
MODULE AirbagController(collision)      LTLSPEC G (collision -> X collision);
  DEFINE                                LTLSPEC G (explode_cmd -> O collision);
    explode_cmd := ...;                 LTLSPEC G (collision -> F explode_cmd);

```

Fig. 13 Examples of contracts attached to modules definitions (upper side) and their straight-forward translation to SMV as LTLSPEC specifications (lower part). Using the contracts annotation, the intent of the formulas (being a pre/post condition) is preserved, when these formulas are translated to SMV their intent is lost.

Contract-based Design. FASTEN also supports contract-based design by providing languages to define interfaces and assemblies. The interfaces can carry contracts (pre-/postconditions) and their compatibility and refinement can be checked by reducing them to SMV. If the verification fails, NuSMV gives a witness (aka. counterexample) which represents a set of values for input ports which lead to the property violation. Once the counterexample is produced, it is lifted at the abstraction level of the DSL and values of various ports can be projected on the diagram. In Figure 14-left we illustrate an example of interface definitions and an assembly for modeling a simplified airbag system containing an instance of a `SensorPlausibilisation` and an instance of `AirbagController`. The semantics of these interfaces and of the assembly is expressed via contracts (the `pre` and `post` in the code). When the verification is run, the results are lifted to the abstraction level of the input DSL – Figure 14-right. When counterexamples are long and complex containing hundreds of entries, they are difficult to understand. We address this by allowing different visualizations of counterexamples (such as Message-Sequence-Charts - Figure 14-right-bottom) or by projecting the values directly in the editor – e.g. the values are presented with a red background in Figure 14-left.

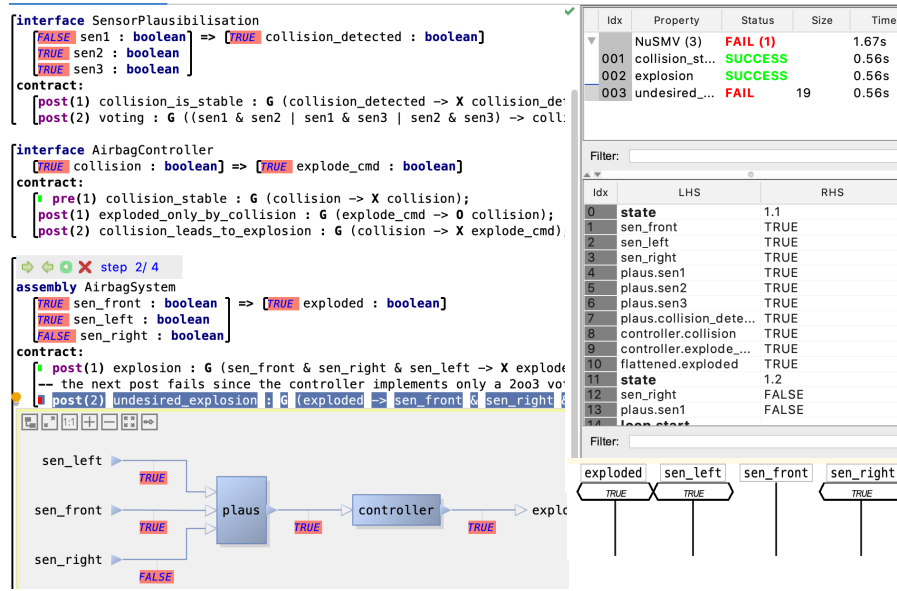


Fig. 14 Examples of the definition of architectures via semantically rich interfaces and attached contracts. Once the contracts are defined, semantic compatibility and refinement can be checked using NuSMV. If a contract fails, the counterexample can be projected in the editor, or selected variables displayed as a Message-Sequence-Chart (bottom-right).

5 Modeling Safety Aspects

When developing systems operating in safety-critical applications, safety standards such as ISO 26262 [16] in the automotive domain prescribe the execution of a safety lifecycle. A safety lifecycle describes a set of activities that should be performed to support the safety assurance of a system, such as 1) hazards and risks analysis and the definition of safety goals, 2) development of functional and technical safety concepts, and 3) verification and validation that the system meets the defined safety goals. Further, the corresponding outputs of these activities, namely the safety work products are also described. The execution of the safety lifecycle is today mostly a manual process, and therefore time consuming and resource demanding. This hinders the development of safety-critical systems in an agile way. This is especially problematic now when, given the current speed of technology advancements, the move towards agile development becomes a necessity [6].

Agile development of safety critical systems calls for automation [39]. To enable automated checks on the safety work products generated by the execution of the safety lifecycle, these work products must be formally defined. We tackle this by defining a set of DSLs, called *FASTEN.safe* [5], that capture safety work products. To achieve a high degree of automation, the safety models have to be deeply integrated with the system models such as requirements and design. In *FASTEN.safe* we experiment with the integration between safety and system engineering models.

In Figure 15 we present an overview of the current set of DSLs for modeling safety cases and outputs of safety analysis (e.g., Hazards and Risk Analysis (HARA), or Systems Theoretic Process Analysis (STPA)). In its most basic usage scenario, *FASTEN.safe* can be seen as a front-end for modeling safety work products like hazards, control structures using STPA or safety cases using the Goal Structuring Notation (GSN) [32]. To support automatic checks on safety case models, we extend the safety languages with semantically rich specializations of GSN constructs which are at higher abstraction level and which are integrated with other system models and verification engines. In Figure 15 we highlight in blue the parts that belong to *FASTEN.safe*. Specialized GSN constructs reference system models such as requirements specifications or architectures annotated with contracts.

DSLs Extensions	Extensions of GSN which Integrate Formal Models					
	SMV-based Req. Spec.	Contract-based Design	Patterns			
Core	Requirements	SMV	Architecture	HARA	STPA	GSN

Fig. 15 Overview of the *FASTEN.safe* stack of DSLs. In blue we highlight the parts that belong to *FASTEN.safe* – they comprise models to support the execution of HARA and STPA, an implementation of GSN language and a set of GSN patterns. Specialized GSN constructs reference formally specified elements of system models such as requirements specifications written with the SMV language or contracts annotations on architecture and components.

5.1 Modeling Support for Hazards and Risks Analysis

Modeling Accidents and Hazards. The objective of the hazard analysis and risk assessment phase in the safety lifecycle is to identify and categorize the hazards of the *system under assurance* (also known as 'item'). From each of the identified hazards, goals for its prevention or mitigation are derived. A hazard is defined as a state or set of conditions of a system that, together with other conditions in the environment of the system, could lead to an accident. An accident is an undesired or unplanned event that results in a loss, including loss of human life or human injury, property damage, environmental pollution, or mission loss [19]. As such, before starting a safety analysis, its purpose should be decided by answering the question "What kinds of losses (also known as "accidents") will the analysis aim to prevent?" [24]. In *FASTEN.safe*, those losses are modeled explicitly, each loss having an identifier and a name (see Figure 16-top). The safety engineer can also model the hazards list, also with an identifier and a name (see Figure 16-bottom). Further, each hazard may be traced to one or more losses. To support the specification of hazards for systems that must demonstrate compliance to ISO 26262, the safety engineer can add ISO 26262-specific properties, i.e., severity, probability of exposure and estimation of controllability.

Losses model

Loss ID	Loss Name
L01	Loss of life or serious injury to people
L02	Electrical damage (economic loss)

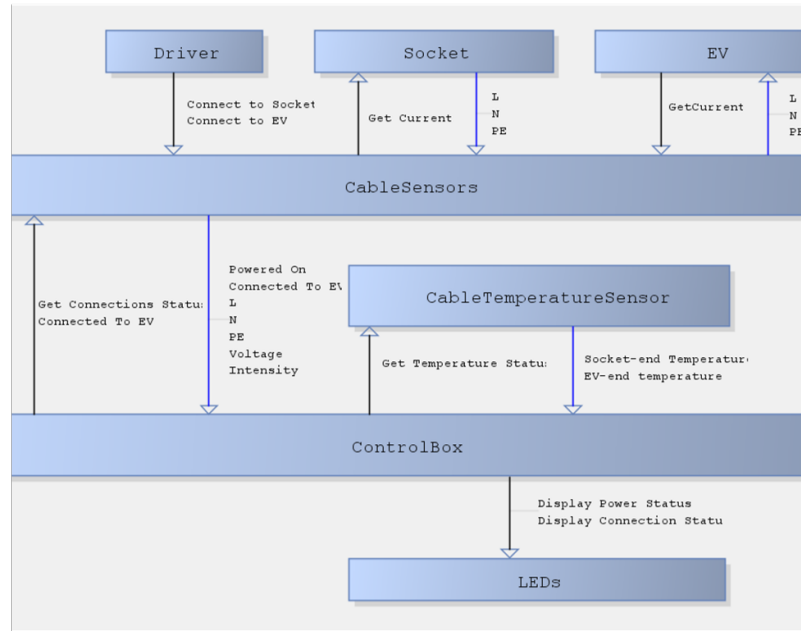
Hazards model

Hazard ID	Hazard Name	Severity	Exposure	Controllability	Associated loss
H01	Electrical hazard	S2	E2	C2	L01
H02	Explosion of the car	S2	E3	C2	L01
H03	Cable temperature too high	S0	E0	C1	L02
H04	The cable catches fire	S1	E2	C2	L02
H05	Smoke comes out of the cable	S1	E1	C1	L02
H06	Tripping	S2	E2	C1	L01

Fig. 16 Models of possible losses (accidents) due to malfunctioning of a charging cable for electric vehicles (top), and models of hazards and their connection to losses (bottom).

Modeling Support for STPA. There are several techniques that can be used for a systematic identification of the hazards and safety goals of the item. The one supported by *FASTEN.safe* is the System-Theoretic Process Analysis (STPA) [24]. After identifying the system-level hazards, a safety engineer identifies the operational and functional hazards. To this end, STPA recommends modeling a control structure, which is defined as a system model that is composed of feedback control loops. An effective control structure enforces constraints on the behavior of the overall system. A controller may provide control actions to its actuators or to a controlled process. Further, a controller may receive feedback from its sensors or from another process

STPA model – Control Structure



STPA model – Unsafe Control Actions

Source Controller	Action	Not Providing Causes Hazard	Providing Causes Hazard
CableSensors	Voltage	N/A	The @controller (CableSensors) provide a voltage outside the expected range while the ControlBox is powered on

Fig. 17 Models of the control structure for a charging cable for electric vehicles (top) and the analysis of possible unsafe control actions (bottom).

(see Figure 17-top). Based on such a control structure, during STPA, safety engineers can specify unsafe control actions. An unsafe control action (UCA) is an action of one of the controllers that, in a particular context and worst-case environment, will lead to a hazard (see Figure 17-bottom). In *FASTEN.safe*, the system engineer can model UCAs as tables, as proposed in the STPA Handbook[24]. Each UCA shall be traced to at least one system-level hazard.

Modeling Safety Requirements. Based on the identified hazards, safety requirements are then specified. In *FASTEN.safe* we extended the requirements base-language presented in Section 3 with safety requirements, which have the particularity that they must trace to one or more hazards (see Figure 18).

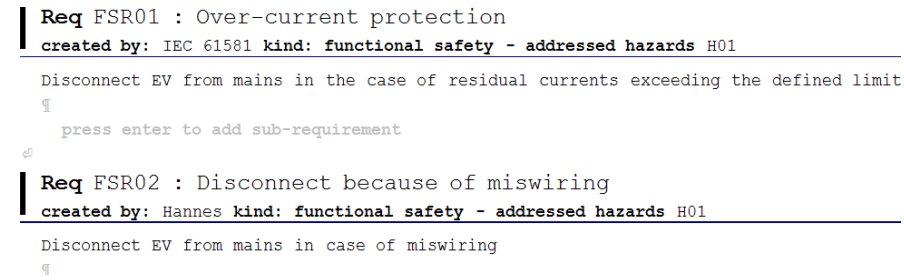


Fig. 18 Examples of safety requirements defined for a charging cable for electric vehicles that are linked to already defined hazards.

5.2 Modeling Safety Cases

A safety case is an argumentation about the satisfaction of system safety requirements based on all the outputs of the safety lifecycle (i.e., assurance artifacts). In *FASTEN.safe*, engineers can model safety cases using GSN and higher level extensions. GSN contains a small number of constructs to capture an argument. It supports the graphical representation of the logical flow between safety claims, depicted as *goals*, *strategies* for decomposition of the claims in sub-claims and evidence for the truth of the claims, depicted as *solution* elements. GSN can also represent contextual information via *context* and *assumption* elements as well as the rationale behind the arguments with the help of *justification* elements. Further, *FASTEN.safe* supports the creation and usage of safety case patterns. Similar to design patterns, safety case patterns describe solutions for how to argue the satisfaction of certain system properties, in a certain context, based on certain type of evidence. Patterns are described as templates entailing placeholders for system-specific information, which have to be filled when the pattern is instantiated in a safety case model.

GSN provides rules for connecting GSN elements among each other in order to obtain a syntactically correct argumentation. However, there are no rules defined for ensuring the semantic validity of an argumentation. In *FASTEN.safe* we explore ways in which safety cases can be made checkable, yet easy to understand by practitioners. To this end, we define specialized GSN constructs that may be connected to each other only via special types of connections. A set of interconnected high-level safety case constructs forms a checkable safety case fragment. The use of semantically rich specialized constructs, integrated with other engineering models, allows for three types of automated checks: 1) intrinsic consistency of safety case models, i.e., checks regarding the validity of the argumentation structure, 2) consistency checks between safety case and system models and 3) verification of safety claims within safety case models by using external verification tools. In the following paragraphs we present these categories of checks in more detail, together with small examples.

1) Intrinsic Consistency. The fact that specialized GSN constructs may only be connected to each other via specialized GSN connections constrains the argumentation structure so that only valid arguments may be constructed. For example, in

Figure 19, an instantiation of the *Argument over hazards* checkable pattern is presented. The *Argument over Hazards Strategy* references a hazard list. This type of strategy can only be supported by goals of type *Hazard Mitigation Goal*, *Eliminated Hazard Goal*, *Negligible Hazard Goal* and *Hazard Substitution Goal*, which reference individual hazards. Once such a goal supports an *Argument over Hazards Strategy* referencing a certain list of hazards, the goal can only reference hazards from that particular list, implemented via (static) scoping rules. If not all hazards of the list referenced by the strategy are referenced by refining goals, then an error is displayed in the editor signaling that the argument is incomplete.

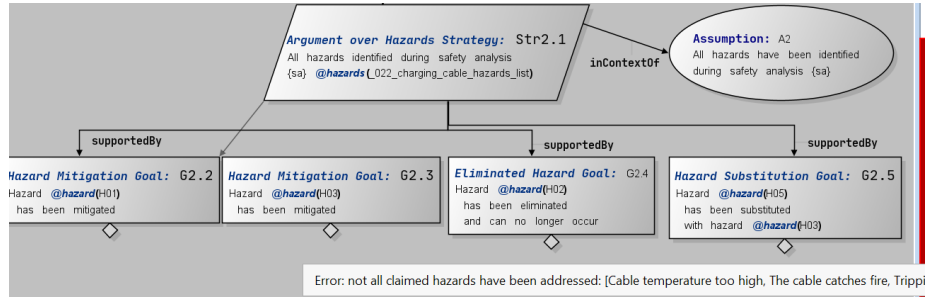


Fig. 19 Instantiation of the *Argument over hazards* checkable pattern. A special strategy called *Argument over Hazards Strategy* references a list of hazards and can only be supported by specialized goals (e.g., *Hazard Mitigation Goals*) that reference hazards from the list. If not all hazards in the list have a corresponding goal, then an intrinsic check will fail and an error is displayed.

2) Consistency with System Models. The integration of specialized constructs extending GSN elements with different types of system models (e.g., hazards, requirements, architecture models) created in the same tool allows safety case models to be aware of changes in the referenced models. Further, we annotate each specialized construct with consistency criteria, which enables automatic identification of inconsistencies between models. For example, given an addition or deletion of an item in the requirements document referenced by the *Argument over Requirements Strategy*, an error will be triggered in the safety case model (see Figure 20). If a new requirement is added, the consistency checks will announce that there is no argument about the satisfaction of the newly added requirement (i.e., the *Argument over Requirements Strategy* is not supported by a *Satisfied Requirement Goal* referencing the newly added requirement). If a requirement is deleted, an error will appear stating that a certain goal references a missing requirement.

3) Verification of Safety Claims. Specialized Solutions can be connected with verification tools and these verification tools can be started directly from within FASTEN. These solutions automatically integrate new verification results as evidence whenever the evidence needs to be updated. The verification results are lifted up to the safety case level, the solution entities being aware if the verification has been successful or not. Further, given a change in the verified system, the evidence is annotated as outdated (see Figure 20), prompting the user to rerun the verification.

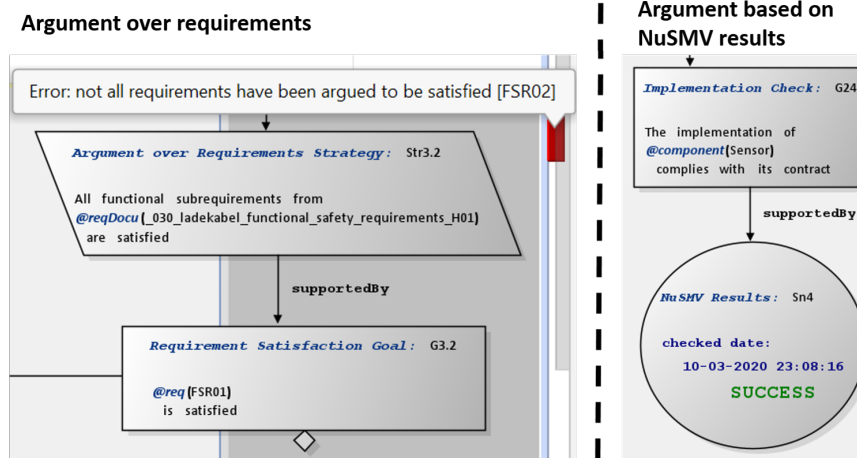


Fig. 20 On the left-hand side it is depicted the instantiation of the *Argument over requirements* checkable pattern for arguing about the satisfaction of all safety requirements identified for the mitigation of hazard **H1** of the charging cable system. A special strategy called *Argument over Requirements Strategy* references a list of requirements and can be linked only to goals of type *Requirement Satisfaction Goal*, which reference individual requirements. If not all requirements have a corresponding goal, then a consistency check will fail and an error is displayed in the editor. On the right-hand side an instantiation of the *Argument based on NuSMV results* checkable pattern is presented. The specialized solution lifts up the NuSMV results, so that it describes that the verification has been successful, but the verification results are outdated.

6 Experiences from the Automotive Domain

An internal research project at Corporate Research of Robert Bosch GmbH works on providing tailored safety-oriented model-based systems engineering solutions to handle incomplete and imprecise specifications for safe autonomous systems. The project builds on the FASTEN platform as a research vehicle to drive toward this vision. In particular, we leveraged FASTEN to develop DSLs that allow the definition of safety-relevant specification as contracts on top of a system model. In the context of this chapter, the project serves as an example and evaluation of the approach proposed by FASTEN. Our solution must be applicable to industrial use cases from the various business units of Bosch. This required that the tool is built around models specified in the OMG Systems Modeling Language (SysML), which we have implemented in MPS [25] (Fig. 21, green). Our extensions to FASTEN are depicted as blue boxes in Fig. 21. They include:

1. Translation of the SysML language to the according SMV extensions (Sect. 4), especially contracts-based design (CBD)
2. A contract pattern language according to [2, 20] as another extension of the CBD extensions
3. Domain-specific extensions to SysML port and contracts to be able to verify certain properties specific to and relevant for some business units

These extensions allowed prototyping of different, use-case specific contract *dialects* and notations that we will introduce in the following. All of them get transformed to FASTEN’s SMV extensions and verified with NuSMV. The FASTEN platform together with our extensions is made available as standalone RCP client, referred to as *SASOCS Workbench* in the following.

6.1 System Model Specification in SysML

We use our SysML MPS implementation described in [26, 25] to specify the system model. In addition, we created a contracts language following the contract definition by [8, 3] that allows specification of formal assumptions and guarantees for SysML blocks in the form of SMV, see Fig. 22.

The SysML system model and the contracts can be translated to the CBD extensions introduced in Sect. 4. Generators translate architectural elements like SysML blocks to their corresponding SMV extensions like component interfaces and assemblies; ports with complex types are translated to multiple simple-typed ports of the SMV extension. Additional generators translate our contract specifications to the corresponding SMV extensions such as pre-conditions (our assumptions) and post-conditions (our guarantees). These translations are lightweight and are therefore performed instantaneously and continuously as the user edits the model with the help of the Shadow Models transformation engine [36].

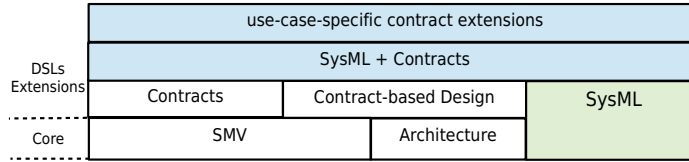


Fig. 21 Bosch-specific extensions to FASTEN (blue), based on SysML (green) and the FASTEN SMV extensions.

```

block Primary_Device {
  port has_data_in flows in boolean
  port torque_request_from_sd flows in boolean
  port has_power_in flows in boolean
  port torque_out flows out boolean
  port torque_request_to_sd flows out boolean

  contract InSlaveNominal
    assume true
    guarantee globally [(self.state == Slave &&
      self.has_power_in &&
      self.torque_request_from_sd)
      => next [(self.state == Slave && self.torque_out)]]

```

Fig. 22 Textual definition of a SysML block with one of its contracts from our EPS use case introduced in Sect. 6.3.

6.2 Contract Specification Patterns

To enhance writability, readability, as well as adoption by practitioners, we allow expression of the contract's assumptions and guarantees in structured English. Our language extension follows the specification patterns by Autili et al. [2]. These patterns are already in use and known within Bosch business units (see Post et al. [28]) which eases adoption of the *SASOCS Workbench*. The top contract in Fig. 23 shows an example of a guarantee formulated as a pattern. The grammar for these patterns is well-specified [2, 20], as is their translation to NuSMV. This made implementation in MPS fairly straight-forward.

```
contract safety
  assume true
  guarantee Globally, it is always the case that
    if self.pedestrian_lights == WALK holds then
      self.traffic_lights == RED holds as well
safety contract safety
  assume true (hard, invariant, always has to be true)
  under the condition (weak assumption) self.pedestrian_lights == WALK
  guarantee self.traffic_lights == RED
guarantee: (self.pedestrian_lights == WALK => self.traffic_lights == RED)
```

Fig. 23 Two contract specifications with similar semantics, but different notation. The *safety contract* on the bottom explicitly distinguishes between *strong* assumptions and *weak* assumptions. The light gray text shows its translation to the *classical* contract notation.

A further separate language extension provides a different notation of safety contracts that is closer to domain experts, i.e., an explicit distinction of *strong* assumptions and *weak* assumptions as suggested by Kaiser et al.: "*Weak assumptions are used to describe a set of possible environments in which the component guarantees different behaviors. This separation is only methodological, and does not affect the semantics of the definition of the original contracts*" [17, Sect. 2.2]. The second contract in Fig. 23 shows an example, how our safety contracts look in the notation proposed by Kaiser.

6.3 Case Studies

We highlight two industrial case studies built in cooperation with Bosch business units: a redundant steering system for highly-automated driving and a map-based extension to automatic cruise control (ACC).

Electronic Power Steering Electronic power steering (EPS) is used in highly-automated driving vehicles. It supports the driver in steering and also can steer the vehicle without any input from the driver by receiving steering commands from a vehicle bus. Hence, the EPS system has high safety, reliability, and availability requirements.

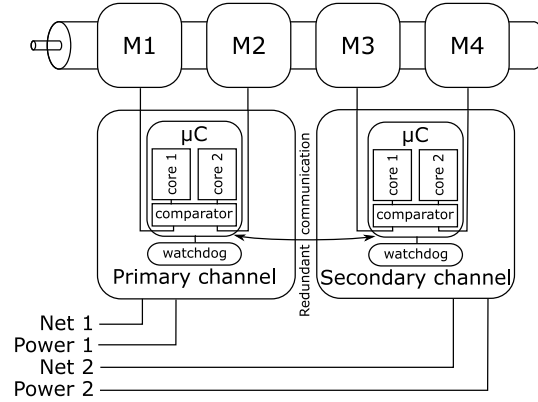


Fig. 24 Schematic overview of the ECU with two redundant channels [4].

We focus only on the electronic control unit (ECU), which is schematically depicted in Fig. 24. It includes two separate channels, named *primary* and *secondary*. Each channel has its own independent power supply and is connected to an independent vehicle bus. Both channels can communicate with each other via redundant intra-ECU communication channels. Each channel contains a lock-stepped microcontroller with an external watchdog and is able to drive two electric motors that actuate the steering. The lock-stepped microcontroller contains two cores that compute the same instructions in parallel. At each cycle, a comparator circuit inside the controller compares the state of both cores. The microcontroller exhibits fail-silent behavior, so in case the two core states are not equal no result is output and the directly connected motors are unlocked. In order to check whether the comparator is working correctly, an external watchdog sends challenges to the comparator and checks the correctness of the response. If the challenge is answered incorrectly or if a timeout error occurs, the entire microcontroller including the comparator is reset while maintaining fail-silent behavior.

Each channel is in one of three modes at any given time: *master*, *slave*, or *passive*. In master mode, the channel calculates the torque for its two motors and sends a request to the other channel in slave mode to set the same torque to its connected motors, so all four motors provide the same torque. If the torque request is not received, the channel in slave mode has to assume that the other channel has failed silently, hence it becomes master and calculates the required torque itself. Since one channel and its directly connected two motors are sufficient to steer the vehicle, the EPS system is still available even if one channel fails. In case a channel does not receive any input from its connected vehicle bus or it detects an internal failure, it switches to passive mode and exhibits fail-silent behavior.

A qualitative analysis of the EPS case study has been carried out by Abele [1] using a fault tree analysis (FTA). In a fault tree analysis, the atomic *basic events* leading to a *top event* are identified through deductive reasoning. The causes of the top event are broken down iteratively by combining them with logical AND and

OR gates. As a result of the FTA, a *minimum cut set*, i. e., the smallest possible combination of basic events that trigger a top event, can be computed and used to reason about the risk of the top event. The author performed this FTA by manually inspecting all possible states and transitions. He demonstrated that the order of events causing the mode transitions can be neglected and thus an FTA is applicable. However, such manual analysis is error-prone and does not scale up when additional channels or faults must be considered.

Bozzano et al. [4] propose a new formal method, based on minimal cut sets, to generate explanations for operational mode transitions, in terms of causes defined as combinations of basic events and recovery actions. This method was applied on the same EPS system and showed its ability to replace the manual analysis. However, the system was modeled in plain-text SMV code and command-line tools that require lots of expert knowledge to be used correctly were applied.

While we have not yet included the analysis method proposed by Bozzano et al. [4] in our tool, we were able to model the system and analyze its behavior for manually injected basic events. In order to do so, the contract pattern language had to be extended to be able to express the order of events. Five additional operators were added to the pattern language introduced in Sect. 6.2, which can be easily translated to the corresponding SMV expressions:

- X The '*next*' operator
- Y The '*yesterday*' operator is the temporal dual of X and refers to the previous time instant
- Z The Z operator is similar to the Y operator, only differing in the way the initial time instant is dealt with: at time zero, Y_φ is false, while Z_φ is true
- O The '*once*' operator is the temporal dual of F (sometimes in the future), so O_φ is true iff φ is true at some past time instant (including the present time).
- H The '*historically*' operator is the past-time version of G (always in the future), so that H_φ is true iff φ is always true in the past

Fig. 22 shows an excerpt of the SysML blocks modeling the primary channel including its ports and the internal attribute indicating the state of the channel. The contract shown in Fig. 22 describes that the primary channel starts in master mode and stays in this mode unless energy or input data is lost. Similar contracts describe the behavior in other modes, the secondary channel as well as the components. Using FASTEN's contract based design extensions (Sect. 4), we first verified that the modeled behavior is indeed correct, i. e., the primary channel stays in master mode indefinitely. Next, we modified the contracts of individual components to manually inject faults, e. g., the power supply does not provide power at some point in time (*finally*). In a contract assigned to the subsuming component (the EPS block), we specified that one channel in master mode is sufficient for the system to function correctly. With the contract refinement check, we were able to check that this specification is indeed valid even if one of the power supplies fails. By manually injecting more faults in other components, e. g., modify the contract of a network connection to eventually not deliver data anymore, we finally reached a point where the contract of the subsuming component is not valid anymore. In other words, we injected a (not

necessarily minimal) cut set of basic events. In order to automate this fault injection process, we plan to integrate approaches such as the method presented by Bozzano et al. [4] into the FASTEN framework, which allows us to specify a fault model externally and automatically extend the system model by these faults. Furthermore, the integration with FASTEN.Safe (Sect. 5), especially the assurance case in Goal Structuring Notation, is on our roadmap.

Automatic Cruise Control (ACC) Extension The purpose of the Automatic Cruise Control (ACC) Extension system is to optimize the energy consumption of a vehicle by operating its powertrain at best efficiency based on additional navigation data. For example, the vehicle is allowed to pick up a bit of speed when driving downhill and use the resulting energy to use less power when driving the following uphill section. The required road geometry is taken from map data and includes road types, slopes, curvature and speed limits for the most probable path ahead.

```

safetyPort Curvature flows in boolean ASIL = ASIL-A
                                FTTI = 500ms
                                cycleTime = 100ms
                                valueRange = [0|1023]
                                End2EndProtection = true

```

Fig. 25 *Safety Port* extension to SysML, allowing additional safety properties to be used within contracts [30].

Within this use-case, we want to verify the component integration based on a model of safety properties of their interfaces. To allow verification with the FASTEN framework, the SysML language was extended to support the specification of safety-relevant properties per port [30], see Fig. 25. These parameters include the automotive safety integrity level (ASIL), the fault-tolerant time interval (FTTI), cycle times of periodic real-time tasks, value ranges, and end-to-end protection being available or not. The chosen properties aim at having a wide range of data types and also represent important characteristics for vehicle software development, so that the approach is proven to be able to cope with the widely used data types found in power train development. These properties can be referred to in a domain-specific contract pattern extension that is translated to FASTEN SMV extensions as well. Thereby, these safety properties are included in the model checking process. Fig. 26 shows an exemplary contract that specifies the relation of input port cycle times and output port cycle times.

```

Contract    EMO_cycleTime
assume      It is always the case, that the property cycleTime of all Input-Ports
              is greater equal than the property of the connected Output-Port.
guarantee    It is always the case, that specified Property cycleTime of all Output-
              Ports is guaranteed.

```

Fig. 26 Case-study-specific contract notation that refers to the safety properties of the introduced safety ports [30].

7 Discussion and Lessons Learned

We started with the FASTEN project by focusing solely on extensions of SMV in October 2017 and then we gradually extended to other verification engines. Bosch started their use in late 2018, and researchers from fortiss actively contributed to FASTEN since late 2019. We estimate the total language development effort to roughly 3 person years, split mainly across the three organizations. In addition to the language development, considerable effort was invested to come up with the present set of abstractions by continuously learning from our interactions with practitioners and the research community. FASTEN is still actively developed and subject to significant extensions. In the following, we present a discussion and summarize the lessons learned from developing and using FASTEN in our research and technology transfer projects.

7.1 Discussion

Maturity. FASTEN is a research tool. Different functionalities are at various Technology Readiness Levels (TRLs) spanning from TRL4 (*validated in a lab*) to TRL6 (*demonstrated in industrially relevant environment*). FASTEN is currently not intended to be used in production. Our main audience are tool builders, tech leads, technology scouts and people responsible for processes, methods and tools who are looking for ways to improve on the status quo and address new challenges caused by the complexity increase of the products. Domain experts have the chance to test hands-on how different aspects of their systems could be modeled, tools builders get inspired about new tooling functions and how they can be integrated, and technology scouts get to know how DSLs can help increase the automation degree of the safety critical systems development.

Extensibility. The FASTEN approach relies on development and integration of modeling languages, in a bottom-up fashion by using stacks of DSLs. The multitude of DSLs presented in this chapter (developed at Siemens, Bosch and Fortiss) show that it is feasible to create domain specific environments on top of FASTEN. As shown in Section 6, the FASTEN framework enables rapid prototyping for the integration of domain-specific languages with formal methods. Its modularity and extensibility allows to quickly connect other DSLs such as SysML. We could easily experiment with different language dialects for representing architecture and contracts.

Tool-driven Research Transfer. The set of functionalities provided by FASTEN exceeds the state-of-practice technologies used in the industry: today's practice is dominated by loosely coupled tool chains. Each aspect from FASTEN (requirements, architecture, design, safety analyses, safety argumentation, verification) is covered by one or more tools, each providing informal specification and modeling means (plain text for requirements, SysML for design or spreadsheets and plain text for safety analyses and argumentation). The loosely coupling of tools lead to information loss

at the boundaries and introduce accidental complexity; their informal content is preventing automation.

We have extensively used FASTEN for interacting with systems and safety tech leads to demonstrate advanced concepts related to model-driven engineering, model-based safety engineering and formal methods. FASTEN allowed us to create demos and verify the usefulness of different modeling and specification approaches for concrete industrial problems. Being able to play with models and analyze them directly in the tool has shown to be extremely useful when presenting to respective business units and other stakeholders.

7.2 MPS Features Supporting Our Work

MPS is a key enabler for the development of FASTEN as we discuss in the following.

Language Development Productivity. MPS enables highly efficient definition, extension and refactoring of languages. The time taken between an idea and its implementation as DSLs and subsequent creation of user-level models is very short (often only a few hours or days), which allows us to perform many iterations over a short timespan. This, in turn, enabled us to evolve the languages based on practitioner feedback and to experiment with new modeling concepts or their combination. In the end this stimulates co-creation of tooling hand-in-hand with domain experts or fellow researchers.

Support for Modular, Extensible and Stackable DSLs. MPS' mechanisms for language modularization proved to be essential for our project, because they allow independent creation of DSLs by different organizations. For each of the integrated tools we have implemented its input language as an MPS language. The extensions are "grown" as stacks of DSLs in a modular fashion with higher-level abstractions, similar to what mbeddr does for the C base language [37]. DSLs addressing different aspects of safety critical systems' development (i.e. requirements, architecture, design, safety engineering) are integrated with each other and seamless workflows beyond the boundaries of single disciplines are enabled.

Notation Freedom. In FASTEN we heavily use combinations of notations: textual, diagrammatic, tree or tabular. MPS allows easy definition of editors and provides multiple notations for the same language concepts, drastically improving usability. We learned that domain-specific or even application-specific notations are key for tool adoption by domain experts, e. g., safety engineers [25]. Sometimes this might even require replicating the look-and-feel of established tools in shape and color to ease adoption.

Editor Automation and Auto-completion. MPS provides several means for increasing the automation of model authoring – context-sensitive auto-completion being the most important. This reduces to some extent the effort of learning new syntax of the verification tools – when auto-completion is used, many gotchas can be avoided.

Syntax-driven Editing and In-editor Errors. The projectional editor of MPS guides the users to create meaningful models and prevents them up-front to make mistakes. With MPS it is easily possible to define extensible sets of context sensitive constraints and display errors in the editor when they are violated. Users get immediate feedback about errors and thereby many inconsistencies can be fixed right away, allowing domain experts to focus on essential things.

Model Annotations. We have used nodes attributes to annotate design models with information of variables values (e.g. the values of counterexamples are projected in the IDE as illustrated in Figure 14). This proves to be very useful when users need to debug their models.

7.3 Open Challenges with MPS-based Tooling

Projectional Editing. While MPS' projectional editing allows maintaining different, domain- and/or stakeholder-specific notations, the projectional editor comes with its own challenges. When used with the expectation of a classical text editor or graphical editor, the resulting editing experience might lead to a lot of frustration when editors are not designed with great care and significant effort. Specialized DSLs like the "grammar cells"[38] ease the creation of editors and partially increase their usability by offering a behavior closer to the textual editors. Despite this, the users still need to be aware that they are not working with a classical textual editor.

IDE Errors Hard to Understand. There is a quite steep learning curve for non-programmers to get accustomed with MPS. Many errors of the IDE seem cryptical to domain experts. Several of these errors are not even meant to be seen by non-programmers and thereby they easily get confused. We consider these situations to be bugs in MPS.

IDE Footprint. The sheer size and resource-consumption of the IDE also tends to hinder adoption. It is hard to argue why a >500 MB IDE is the right choice for working with small domain-specific artifacts that may sometimes look like simple text snippets. Developing a lightweight MPS-based IDEs is still an open challenge. Recently, there has been highly promising work done to deploy MPS on a server and access its models via web browsers in the *modelix*⁶ project. We plan to leverage on this in order to make our DSLs more accessible by occasional users and thereby make experimentation by domain experts easier.

Deployment. While MPS does provide support for deployment of languages as plugins or standalone IDE, this support is still fragile and requires patching jars for advanced customizations. Furthermore, integration into CI pipelines requires handling of a significant technology stack (such as ant, maven, and gradle) – automation and maintenance of the builds remains challenging.

⁶ www.modelix.org

8 Conclusions and Future Work

In order to tackle the complexity of safety critical systems, increase trust and enable agile development we need semantically rich and deeply integrated models about different aspects of the system from requirements, design and safety engineering. Currently, the industry is using ad-hoc and loosely coupled tool chains, featuring informal models, and they cannot cope with today's challenges. In this chapter we presented FASTEN, an extensible platform based on JetBrains MPS, developed and used by researchers from three organizations over the last three years, both from industry and academia. We presented a set of requirements DSLs that enable the transition from informal requirements to formal models, DSLs for the formal specification of architecture and system-level behavior, and DSLs for safety engineering and assurance. We successfully used the DSLs to model two industrial systems from the automotive domain.

MPS is a key enabling technology for the FASTEN system. *On tools building side*, MPS empowered us to efficiently build extensible stacks of DSLs; to integrate independently developed modular DSLs; to provide most appropriate notations; to equip language constructs with rich and extensible sets of semantic rules; to implement advanced editing support for creating models; and, last but not least, to integrate external analyses tools and present the analyses results at the abstraction level of the DSL. *On research and technology transfer side*, MPS enabled us to experiment with adequate abstractions, to prototype new ideas in closed-loop interactions with domain experts or fellow researchers, as well as to cooperate beyond the borders of a single company. FASTEN is an open-source and open-innovation platform for research and technology transfer in the field of safety critical systems.

Future Work. FASTEN can be easily extended with new DSLs in order to experiment with higher-level modeling abstractions. FASTEN is still under development both at the platform level as well as regarding the higher-level modeling languages. We are extending and fine-tuning the DSLs based on the feedback from domain experts such as requirements, system or safety engineers. We plan future work along three directions: enabling new modeling abstractions; integration of existing abstractions to enable higher-level workflows, and better integration of tooling.

Examples for directions for modeling extensions are boilerplate patterns for the specification of timing/reliability aspects of requirements, integration of failure models and formal models for robustness analysis of the design, or enriching the semantics of safety argument structures to enable more automated checks. Furthermore, we plan to integrate model-based fault injection approaches similar to [4].

Regarding directions for modeling integrations, we are looking at the integration of the modeling languages developed by Bosch with more functionality from FASTEN.Safe, especially the assurance case in GSN; or the integration of system models with the software or hardware aspects.

Finally, examples of better integration of tooling in modeling workflows is to further improve the lifting of analysis results, to enhance the existing interaction with analysis engines and to integrate new engines.

References

1. Abele, A.: Transformation of a state description into a qualitative fault tree. In: Praxisforum Fehlerbaumanalyse & Co. (2019)
2. Autili, M., Grunske, L., Lumpe, M., Pelliccione, P., Tang, A.: Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar. *IEEE Trans. Software Eng.* **41**(7), 620–638 (2015)
3. Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T., Larsen, K.G.: Contracts for systems design: Theory. Tech. rep., INRIA (2015)
4. Bozzano, M., Munk, P., Schweizer, M., Tonetta, S., Vozárová, V.: Model-based safety analysis of mode transitions. In: Proc. of SAFECOMP (2020)
5. Cărlan, C., Ratiu, D.: FASTEN.Safe: A model-driven engineering tool to experiment with checkable assurance cases. In: Proceedings of the International Conference on Computer Safety, Reliability, and Security (SAFECOMP), *LNCS*, vol. 12234, pp. 298–306. Springer (2020)
6. Cawley, O., Wang, X., Richardson, I.: Lean/agile software development methodologies in regulated environments - state of the art. In: Proceedings of First International Conference on Lean Enterprise Software and Systems - LESS, *Lecture Notes in Business Information Processing*, vol. 65, pp. 31–36. Springer (2010)
7. Cimatti, A., Clarke, E.M., Giunchiglia, E., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. In: Proceedings of the 14th International Conference on Computer Aided Verification, CAV '02, p. 359–364. Springer-Verlag, Berlin, Heidelberg (2002)
8. Cimatti, A., Tonetta, S.: A property-based proof system for contract-based design. In: 38th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2012, Cesme, Izmir, Turkey, September 5-8, 2012, pp. 21–28 (2012)
9. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004), *Lecture Notes in Computer Science*. Springer (2004)
10. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08, p. 337–340. Springer-Verlag, Berlin, Heidelberg (2008)
11. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st International Conference on Software Engineering, ICSE '99, p. 411–420. Association for Computing Machinery, New York, NY, USA (1999)
12. Erdweg, S., Van Der Storm, T., Völter, M., Boersma, M., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., et al.: The state of the art in language workbenches. In: International Conference on Software Language Engineering, pp. 197–217. Springer (2013)
13. Graydon, P.J.: Formal assurance arguments: A solution in search of a problem? In: 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pp. 517–528 (2015). DOI 10.1109/DSN.2015.28
14. Hatcliff, J., Wassyng, A., Kelly, T., Comar, C., Jones, P.: Certifiably safe software-dependent systems: Challenges and directions. In: Future of Software Engineering Proceedings, FOSE 2014, p. 182–200. Association for Computing Machinery, New York, NY, USA (2014)
15. Holzmann, G.: Spin Model Checker, the: Primer and Reference Manual, first edn. Addison-Wesley Professional (2003)
16. ISO: 26262: Road vehicles-Functional safety, vol. 26262. International Organisation for Standardization (ISO) (2018)
17. Kaiser, B., Weber, R., Oertel, M., Böde, E., Nejad, B.M., Zander, J.: Contract-based design of embedded systems integrating nominal behavior and safety. *Complex Systems Informatics and Modeling Quarterly (CSIMQ)* **4**, 66–91 (2015)

18. Kelly, T., Weaver, R.: The goal structuring notation – a safety argument notation. In: Proc. of Dependable Systems and Networks 2004 Workshop on Assurance Cases (2004)
19. Knight, J.: Fundamentals of dependable computing for software engineers. CRC Press (2012)
20. Konrad, S., Cheng, B.H.C.: Real-time specification patterns. In: 27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA, pp. 372–381 (2005)
21. Kossak, F., Mashkoor, A., Geist, V., Illibauer, C.: Improving the understandability of formal specifications: An experience report. In: C. Salinesi, I. van de Weerd (eds.) Requirements Engineering: Foundation for Software Quality, pp. 184–199. Springer International Publishing, Cham (2014)
22. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: G. Gopalakrishnan, S. Qadeer (eds.) Proc. 23rd International Conference on Computer Aided Verification (CAV’11), *LNCIS*, vol. 6806, pp. 585–591. Springer (2011)
23. Leveson, N.: Engineering a Safer World, first edn. MIT Press (2012)
24. Leveson, N.G., Thomas, J.P.: *Stpa handbook*. Cambridge, MA, USA (2018)
25. Munk, P., Nordmann, A.: Model-based safety assessment with SysML and component fault trees: application and lessons learned. *Software and Systems Modeling* (2020)
26. Nordmann, A., Munk, P.: Lessons learned from model-based safety assessment with SysML and component fault trees. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, *MODELS 2018*, pp. 134–143. ACM (2018)
27. OMG: *OMG Systems Modeling Language (OMG SysML), Version 1.3* (2012). URL <http://www.omg.org/spec/SysML/1.3/>
28. Post, A., Menzel, I., Hoenicke, J., Podelski, A.: Automotive behavioral requirements expressed in a specification pattern system: a case study at bosch. *Requirements Engineering* **17**(1), 19–33 (2012)
29. Ratiu, D., Gario, M., Schoenhaar, H.: FASTEN: An open extensible framework to experiment with formal specification approaches. In: Proceedings of the 7th International Workshop on Formal Methods in Software Engineering, *FormalISE ’19*, p. 41–50. IEEE Press (2019)
30. Rauhut, J.: Safety assurance of open context systems. Master’s thesis, University of Applied Science Esslingen (2020)
31. Spichkova, M., Zamansky, A.: Teaching of formal methods for software engineering. In: Proceedings of the 11th International Conference on Evaluation of Novel Software Approaches to Software Engineering - Volume 1: COLAFORM, (ENASE), pp. 370–376. SciTePress (2016)
32. The Assurance Case Working Group: Goal structuring notation community standard version 2 (2018). URL <https://scsc.uk/scsc-141B>
33. Tommila, T., Pakonen, A.: Controlled natural language requirements in the design and analysis of safety critical i & c systems. Tech. rep., VTT, Finland (2014)
34. Viger, T., Salay, R., Selim, G.M.K., Chechik, M.: Just enough formality in assurance argument structures. In: Computer Safety, Reliability, and Security - 39th International Conference, *SAFECOMP 2020*, Lisbon, Portugal, September 16-18, 2020, Proceedings, Lecture Notes in Computer Science. Springer (2020)
35. Völter, M., Kolb, B., Birken, K., Tomassetti, F., Alff, P., Wiart, L., Wortmann, A., Nordmann, A.: Using language workbenches and domain-specific languages for safety-critical software development. *Software & Systems Modeling* (2018)
36. 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*, p. 61–65. ACM (2019)
37. Voelter, M., Ratiu, D., Kolb, B., Schaetz, B.: mbeddr: Instantiating a language workbench in the embedded software domain. *Automated Software Engineering* **20**(3), 339–390 (2013)
38. Voelter, M., Szabó, T., Lisson, S., Kolb, B., Erdweg, S., Berger, T.: Efficient development of consistent projectional editors using grammar cells. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, *SLE 2016*, p. 28–40. ACM (2016)
39. Vuori, M.: Agile development of safety-critical software. Tech. rep., Tampere University of Technology. Department of Software Systems. Report 14 (2011)