Engineering Science and Technology, an International Journal 23 (2020) 494–506

HOSTED BY

Contents lists available at ScienceDirect

Engineering Science and Technology, an International Journal

journal homepage: www.elsevier.com/locate/jestch

Full Length Article

(User-friendly) formal requirements verification in the context of ISO26262

Denis Makartetskiy [a], Guido Marchetto [a], Riccardo Sisto [a], Fulvio Valenza [a,*], Matteo Virgilio [a], Denise Leri [b], Paolo Denti [b], Roberto Finizio [b]

[a] Dipartimento di Automatica e Informatica, Politecnico di Torino, c.so Duca degli Abruzzi 24, Torino I-10129, Italy
[b] Centro Ricerche Fiat (CRF), Str. Torino 50, Torino I-1004, Italy

ARTICLE INFO

Article history:
Received 19 March 2019
Revised 13 September 2019
Accepted 18 September 2019
Available online 1 October 2019

Keywords:
ISO26262
Formal methods
SysML

ABSTRACT

In order to achieve the highest safety integrity levels, ISO26262 recommends the use of formal methods for various verification activities, throughout the lifecycle of safety-related embedded systems for road vehicles. Since formal methods are known to be difficult to use, one of the main challenges raised by these ISO26262 requirements is to find cost-effective approaches for being compliant with them. This paper proposes an approach for requirements formal verification where formal methods, languages, and tools are only minimally exposed to the user, and are integrated into one of the commonly used system modeling environments based on SysML. This approach does not require particular expertise in formal methods still allowing to apply them. Hence, personnel training costs and development costs should be kept limited. The proposed approach has been implemented as a plugin of the Topcased environment. Although it is limited to discrete system models, it has been successfully experimented on an industrial use case.

1. Introduction

The ever-increasing complexity of automotive systems is raising new challenges in the area of system safety engineering. This statement is confirmed by our everyday life, where we continuously interact with electrical and/or electronic (E/E) systems, whose possible failures could have enormous safety consequences and implications (car airbag and cruise control are just two representative examples). Starting from these considerations, ISO 26262 [1] is a standard that was developed to meet the specific needs of safety critical systems within road vehicles and it applies to a complete automotive safety lifecycle: management, development, production, operation, service, decommissioning. It provides an automotive-specific risk-based approach to determine the Automotive Safety Integrity Levels (ASIL), a risk classification scheme defined in the standard to analyze and classify the impact of a particular potential hazard. The V model described in part 6 of the ISO 26262 standard (Fig. 1) is very similar to a standard V model for software development. The main differences are the initial focus on safety, and the increased number of upwards design phase verification stages. This software V model is included into a broader V model that refers to the whole product development and that is articulated into system, software and hardware development phases.

During the various product development phases, ISO26262 recommends the use of various specification and verification techniques. When dealing with items classified at the highest ASIL levels, formal and semi-formal methods are recommended. The most relevant recommendations about formal methods refer to the left-hand side of the V models, where formal specification is recommended for the specification of safety requirements and system designs at different abstraction levels, while formal verification is recommended for upwards design phase verifications. For example, the compliance and consistency of the software safety requirements with the technical safety requirements (expressing the output requirements of the system level development process) has to be verified, and formal verification is among the recommended techniques to be used for this purpose.

While the key role of formal verification techniques in the context of ISO 26262 is generally well understood, its application continues to appear a bit obscure and people with industrial background are often still reluctant in leveraging formal methods as they are known to be hard-to-use and very time consuming

* Corresponding author at: Dipartimento di Automatica e Informatica, Politecnico di Torino, c.so Duca degli Abruzzi 24, Torino I-10129, Italy.
E-mail address: fulvio.valenza@polito.it (F. Valenza).

Peer review under responsibility of Karabuk University.

https://doi.org/10.1016/j.jestch.2019.09.005
2215-0986/

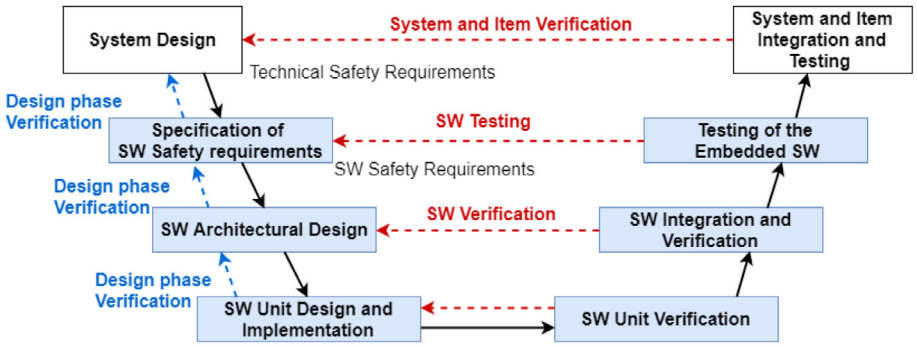


Fig. 1. The ISO-26262 SW V model.

(therefore onerous, from an economic standpoint) [2]. All these observations lead us to the intuition that more user-friendly tools are needed to help people not familiar with formal methods, to approach and profitably use modern tools to formally model and verify the systems they are actually designing and implementing. Another feasible approach could be that of training personnel by spreading the necessary knowledge and skills to autonomously implement and complete verification tasks. While more appealing in the long-term, this solution would not be easily practicable from a cost perspective since it would require training time and associated costs, not to mention the initial learning curve which many companies may not be able to afford.

This paper addresses these issues by proposing techniques and tools that can contribute to remove the main obstacles to the use of formal methods in ISO26262-compliant development processes. Especially, our aim is to aid in the design phase verifications expected in the left-hand side of the V models, where the ability to compare models at varying levels of abstraction (going back up the abstraction levels) is required, and in the requirements-based testing activities that are required in the right-hand side of the V models.

More specifically, our work tries to cope with the above mentioned challenges, by proposing a modeling approach which does not require sophisticated skills in formal methods and is tailored on concepts that industry people are already aware of and familiar with: state machines, activity diagrams and sequence diagrams are palpable examples of "languages" already known to both academic and industry people that can be leveraged to model system behaviors. In particular, these languages are already part of UML-based semi-formal notations, such as for example SysML [3], which is widely known and accepted in the automotive industry [4,5]. A valuable feature of SysML is that it includes the possibility to specify hierarchical requirements with associated behavioral models expressed in the user-friendly languages mentioned above. This enables the management of safety requirements according to the recommendations of ISO26262 and at the same time the formal specification of requirements through such behavioral models. Although SysML is known to be a semi-formal notation, its behavioral notations are quite close to fully formal models, and they can be converted into fully formal models by resolving the few variation points left in the definition of these languages.

Having created formal models of the requirements and the system at various levels of abstraction raises the question of how these models relate to each other. Does a model at a lower level of abstraction actually comply with a higher level one, by adding more detail while preserving the coarser behavior of the higher level one? In order to answer this question in a formal way, it is possible to use refinement checking, i.e. a formal check of whether a more abstract formal artifact is correctly refined into a less abstract one. The main advantage of refinement checking, compared to other formal verification techniques such as model checking, is that it does not require the user to specify formal properties in temporal logics, which is known to be challenging [2], but it can be applied directly to the models developed by the user with the more user-friendly notations. In particular, we propose the application of refinement checking to the behavioral models extracted from a SysML model, which is a novel approach in the ISO26262 context, as discussed in the Related Work Section.

One last ingredient of our work is a technique for automatic generation of test cases from the formal requirements specifications expressed by means of behavioral models. These automatically generated test cases can be used for requirements-based testing, which is one of the testing methods recommended by ISO26262 in the right-hand side of the V models, and also as an alternative to refinement checking for upwards design-time verifications, when refinement checking is not feasible because of the excessive complexity of the formal models. While the automatic generation of test cases from formal behavioral specifications is not new in itself, this way of using it, made possible by the proposed approach is new.

Our focus in this paper is on discrete systems and related requirements. While the approach we are proposing could be extended to deal also with continuous-time systems or requirements involving quantitative time, such extensions are outside the scope of this paper.

The approach we are proposing exploits a number of techniques already developed in the past by other researchers: the extraction of formal models from UML/SysML behavioral diagrams, the verification of refinement on formal models, and the automatic generation of test cases from formal models. Our main contribution is the way they are joined together so as to comply with the recommendations of the ISO26262 standard about the use of formal methods (especially for design-time verification), while at the same time hiding most of the complexity of formal methods from the final user, thus achieving a toolset that can be used even by users without specific expertise in formal methods. While there are alternative proposals for achieving this kind of hiding, none of them achieves all the goals we do.

For demonstrative purposes, our ideas have been developed and integrated, in plugin form, into Topcased, an Eclipse based modeling environment that supports SysML, where all the details related to the formal verification processes are kept, as much as possible, behind the scenes and automatically handled by the tool.[1] It is important to note, however, that the proposed tool-chain is not intended as a full replacement of all ISO-26262-related activities, rather as a possible way to support some of the formal-method-related parts. It can be considered as a core functionality that needs

[1] The plugin is available for download at https://github.com/netgroup-polito/VeTeSS-Topcased-Verification-Plugin.

to be then combined with other toolsets in order to achieve a full ISO-26262-compliant process.

In the rest of the paper we cover all these aspects with the following agenda: Section II presents the proposed way for specifying safety requirements in the SysML context. Section III shows how refinement can be formally verified at design time when developing new safety requirements, according to the recommendations of ISO 26262, while section IV shows the proposed technique for automated generation of tests based on safety requirements. Some results about the application of the proposed approach to an industrial use case are presented in section V. Section VI discusses related work and, finally, section VII concludes the paper.

2. Safety requirements formalization

Formal verification is possible as far as formal specifications are available. In our case, the starting point for obtaining formal specifications is the SysML modeling notation, already in use in vast areas of the automotive industry. SysML is a UML-based semi-formal notation. It inherits from UML several features, and extends UML with other features that are specific for system modeling. One of the key extensions of SysML with respect to UML is the possibility to model requirements and their relationships. In SysML, the system structure is described by means of structural diagrams. The main ones are block diagrams, describing a system or part of it as interconnected blocks. Behavioral diagrams complete system models by allowing the specification of system behavior by means of various formalisms. All behavioral diagrams are inherited from UML: activity diagrams use a formalism similar to Petri nets, sequence diagrams use message sequence charts, state machines are based on hierarchical extended state machines, while use case diagrams are a way to represent interfaces and use cases. Finally, requirement diagrams are a way to represent system requirements and their relationships. Both requirement-to-requirement relationships and relationships between requirements and system model elements are possible. The main relationships that can be defined are:

- The derive relationship relates a derived requirement to one of its source requirements;
- The satisfy relationship relates a model element to a requirement that is intended to be satisfied by that element;
- The refine relationship relates a model element and a requirement where one of the two refines with greater detail the other. The refine relationship allows, among the other things, to associate behavioral models with requirements;
- The verify relationship relates a requirement to a test case that can be used to verify whether the requirement is satisfied.

Note that a requirement can derive from one or more source requirements and more requirements can derive from the same source requirement and a similar consideration holds for the other relationships.

A possible alternative to SysML is EAST-ADL [6], which is a rich framework designed specifically for the automotive systems and with ISO-26262 in mind. As the key language features, we exploit are present in both SysML and EAST-ADL, the method proposed here can be cast to use EAST-ADL instead of SysML. Despite the better specificity of EAST-ADL for automotive systems, it has still less tool support and diffusion. For this reason, we preferred using SysML for our proof of concept.

The way we propose for using SysML modeling for specifying system-level safety requirements in an ISO26262 project is shown in Fig. 2, where the yellow arrows represent derive relationships (e.g. technical safety requirements are derived from functional

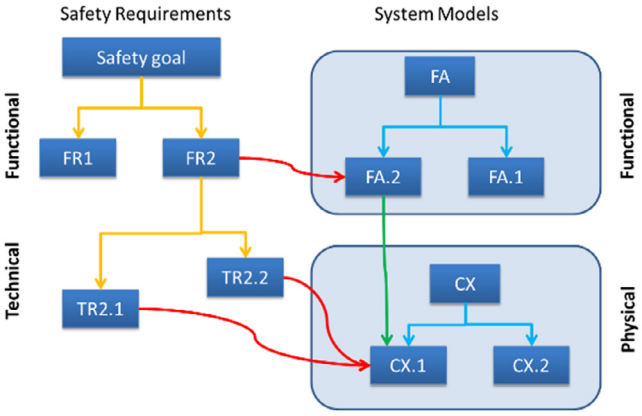


Fig. 2. A possible way of using SysML for specifying ISO26262 requirements.

safety requirements) and red arrows represent satisfy relationships. On the system modeling side, decomposition relationships (the blue ones in Fig. 2) are used for representing the hierarchical decomposition of a system, and refine relationships (green) for connecting corresponding system blocks at different abstraction levels (for example, functional blocks to corresponding physical components). Note that here we propose to use the same refine relationship also used for binding a requirement to its corresponding system element or behavioral model, but to relate system models.

The key point of our proposal is that engineers in charge of defining safety requirements should specify them as SysML requirements, and, for each requirement that needs a formal specification, they have to provide an associated behavioral diagram that specifies the intended behavior of the system according to the requirement.

An example of an airbag requirement specification expressed at high abstraction level is shown in Fig. 3, in its textual form (left) and in the form of a behavioral diagram (state machine) that specifies the expected behavior of the system according to this requirement (right). In the sample of Fig. 3 a state machine with two events is used to express the requirement: start_cc represents the start of a critical collision while deploy represents the deployment of the airbag. The fact that the airbag must not deploy if no critical collision started can be represented by a machine with two states: one state represents the behavior of the system before the start of a critical collision while the second state represents the behavior of the system after that event. The fact that the deploy event is possible only in the second state and not in the first one captures the requirement.

Another example of how a more complex requirement can be modeled is demonstrated in Fig. 4. Starting from such requirements, more detailed requirements can then be developed, in order to specify the technical means by which the original requirements can be satisfied. The new, more detailed, requirements must refine the original abstract ones, and this refinement relationship can be checked, as discussed in the Section 3.

In our view, the behavioral diagram associated with a requirement is its authoritative specification, while the text is just its explanation. We assign a formal semantics to the UML behavioral diagrams associated with SysML requirements by translating them to CSP (Communicating Sequential Processes) [7,8], a language for the formal description of discrete-event sequential processes that run concurrently and that can communicate with one another. This way of assigning formal semantics to UML behavioral models is not new [9]: translation algorithms from behavioral diagrams to CSP already exist that resolve the few variation points left with those

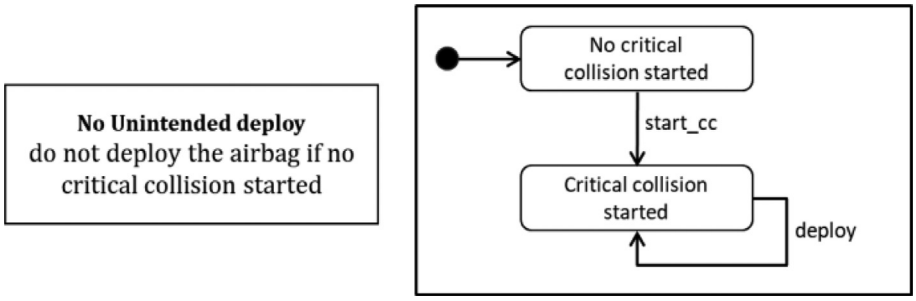


Fig. 3. Requirement modeling example.

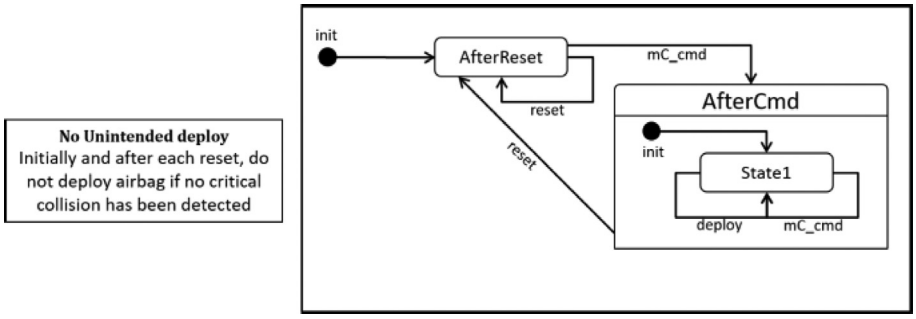


Fig. 4. Another requirement modeling example.

diagrams by the UML semantics (e.g., the way events are dispatched to a state machine). The main limitation of this CSP-based approach is that CSP cannot be used to express quantitative time and continuous-time models (e.g, we cannot express formally the requirement that the time that elapses between two given events is less than 1 ms).

3. Formal verification of safety requirements derivation

3.1. The proposed refinement checking method

The basic idea of the verification approach we propose is that derived requirements have to refine their source requirements. If each requirement is associated with a behavioral model that specifies the intended behavior of the system according to that requirement, the behavioral models of requirements bound by the derive relationship can be compared, in order to check that the behavior of source requirements is correctly refined by the behavior of derived requirements. As in our approach SysML requirements are formally represented by CSP processes, and CSP has a refinement theory, this theory can be exploited to check refinement relationships. The formal verification of refinement relationships is called refinement checking [7] and it is a well-known formal verification technique in CSP.

Generally speaking, a refinement relationship can be any relationship that formalizes the concept of refinement, which ideally should hold between a specification and its implementation. If the specification is a requirement, and its implementation is a model of the system that should fulfil the requirement or a set of (more detailed) derived requirements, refinement checking can be used to verify that the system model or the derived requirements satisfy the original requirement.

For example, *traces refinement* between a more concrete model M and a more abstract one M^A holds if each execution trace (i.e. sequence of events) of M is also an execution trace of M^A. Instead, the relationship does not hold if M can have sequences of events that cannot occur in M^A.

Traces refinement is known to preserve all safety properties, i.e. all properties specifying that something unwanted will never occur. This means that if M is a traces refinement of M^A, then M satisfies all the safety properties satisfied by M^A. Another relationship that could be considered is simulation refinement. M is a simulation refinement of M^A if M simulates M^A step by step, i.e. if each possible step in M always corresponds to a possible step in M^A and the steps in M and in M^A lead to states again related by the same relationship. As it can be argued, simulation refinement is stronger than traces refinement, and traces refinement is implied by simulation refinement. Simulation refinement is known to preserve all linear temporal logic (LTL) properties (not only safety properties) and a large subset of all computational tree logic (CTL*) properties. Often, weakened versions of these relationships are also considered. For example, weak simulation refinement requires that only some of the steps (the observable ones) have a match in the other model. In this way, it is possible to correctly relate cases where one step in the abstract model corresponds to a sequence of steps in the refined model. Of course, in this case only properties that do not involve non-observable steps can be preserved.

As we are mostly interested in safety properties, traces refinement is enough for our purposes. Moreover, as we have to relate models at different levels of abstraction, we use weak traces refinement, which disregards the events classified as unobservable. In our case, events are considered observable only if they belong to the alphabet of the abstract model, while events occurring only in the refined model are considered unobservable, unless they are explicitly declared to match a corresponding different event in the abstract model, in which case the corresponding events are renamed to become the same event. In this way, a refined model is allowed to generate a number of unobservable events between any two observable events matching the abstract model.

As we allow that more requirements derive from an abstract requirement, in general we need to check that a collection of

requirements $R_1,\ldots,R_n$ correctly refines an abstract requirement Ra. The refinement checking problem can be formulated by checking that a CSP process *I* is a weak traces refinement of another CSP process Pa, where Pa represents the system behavior required by Ra and I represents the system behavior jointly required by $R_1,\ldots,R_n$, with the events that do not correspond to events of Pa hidden. Process *I* is computed as $I=(Q_1\backslash a_1)||\ldots(Q_n\backslash a_n)$, where || is the parallel composition operator of CSP, which implies synchronization on all shared events, Qi is the process that represents the system behavior required by R_i, with the events that should be mapped to Pa's events properly renamed, ai is the set of events of Q_i that should be hidden because they have no corresponding event in Pa, and \ai is the operator that hides the events in ai.

3.2. Integrating the proposed method into an ISO 26262 process

Because of its mentioned meaning, refinement checking can be used as a way to perform what ISO 26262 calls design-time verification of compliance. For example, ISO 26262 part 6 clause 6.4.7 specifies that the software safety requirements shall be verified to provide evidence for their compliance and consistency with the technical safety requirements. These design-time verifications can be done using formal methods (formal verification is recommended for the highest ASIL levels), provided the requirements or system models to be compared are formally specified. Let us assume that formal specification of requirements is done using SysML behavioral diagrams, as shown in Section 3.1, possibly adding a specification that some pairs of different events should be considered the same. Having these specifications, formal verification can be done by refinement checking, using weak traces refinement, and the classification of events into observable and unobservable ones as specified above.

A typical development process in the automotive industry is to use SysML for safety requirements specification and for expressing the initial system architecture, Then, a more detailed Simulink system model is developed. In order to extend our approach to perform refinement checking between the requirements and the Simulink model, one should extract a formal CSP model out of the Simulink model. An alternative, enabled by our approach, is to verify refinement by testing. In particular, it is possible to use a form of "back-to-back comparison testing": test cases are derived from the SysML safety requirements and then applied to the Simulink model (after having converted them into a form accepted by Simulink). At a later stage, when C code is generated from the Simulink model, the same test cases can be applied to test the C code, after proper conversion of the involved events. This is a form of requirements-based testing, one of the test methods recommended by ISO26262.

3.3. Implementation of the proposed refinement checking method

Several refinement checkers based on CSP theory exist. In our Topcased plugin, verification is implemented by the tools provided by PAT (Process Analysis Toolkit) [10], which efficiently implements refinement checking according to this theory and also provides algorithms for automatic generation of CSP processes from UML behavioral diagrams.

The procedure followed by the plugin that we implemented is shown in Fig. 5. It works as follows:

- Derive relationships are identified in the model, and state machines (or other behavioral models) associated with the requirements bound by derive relationships are extracted from the model, by an XSLT transformation of the XML representation of the model.

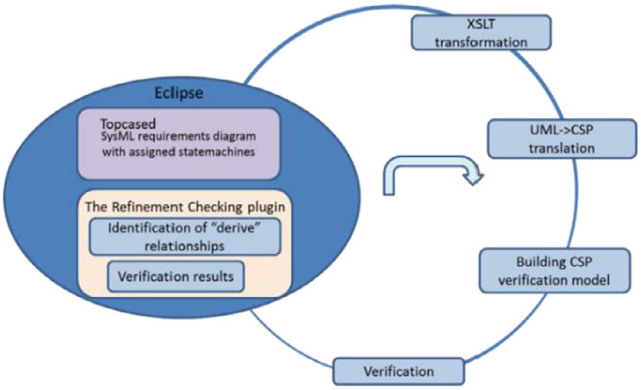


Fig. 5. The refinement plugin verification workflow.

- A corresponding CSP process is automatically generated from each extracted behavioral model. This is done by exploiting the algorithms provided by PAT to create CSP processes from UML behavioral diagrams.
- For each group of behavioral models bound by the derive relationship for which refinement has to be checked, a CSP verification model is built which includes the specification of the weak traces refinement relationship to be checked and of the involved CSP processes, built as explained in Section 3.1.
- PAT is invoked on the CSP verification model in order to perform refinement checking between the corresponding CSP specifications.
- Verification results (including counterexamples in case of errors found) are converted back to the SysML world and reported to the user by the plugin.

Considering refinement checking as part of our formal verification approach, the concrete toolchain needed to run a complete verification task (starting from the modeling phase and ending up with a verification outcome) is depicted in Fig. 6 (the boxes with blue background represent tools specifically developed for our purposes while the other boxes are already existing tools; the part about test case generation is not included in the picture but it will be discussed in Section 4).

As SysML is used for the specification of requirements and their associated behavioral models, any modelling tool that supports SysML and that can export models in XMI format can be used. SysML is also used to express the derivation relationships that bind requirements (in this way the user can specify how new requirements have been derived from the previous ones) and satisfy relationships that bind requirements to the system model elements that implement them. The derivation relationships also enable requirements tracing, which is also required by ISO26262. The information about the mapping of the events occurring in the requirements bound by a derive relationship can be added to the derive relationship itself as an attribute in SysML (it is specified as a collection of event pairs).

The tool chain, which was described preliminarily in [11], is integrated in the Topcased environment where a specialized plugin is used to drive the whole verification process. More precisely, the plugin analyzes the SysML model and automatically extracts derivation relationships binding requirements.

The user can interactively decide for which of such relations refinement should be checked, by selecting the requirements involved in the verification. Then, for each selected derivation relationship, the plugin automatically extracts the behavioral models associated with the requirements bound by the relationship, translates them into CSP, and generates the input for PAT as already

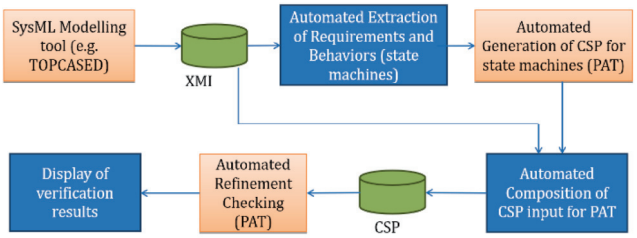


Fig. 6. Formal verification toolchain using state machines and refinement checking.

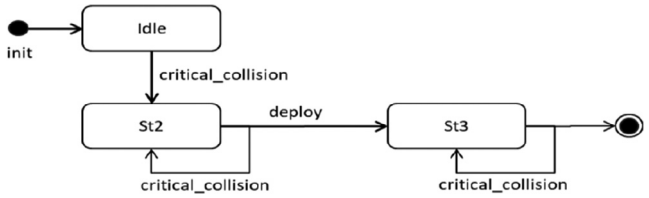


Fig. 8. The state machine attached to the more concrete requirement.

explained. Finally, PAT is automatically invoked and refinement is checked. The results (including counterexamples in case of failure) are finally reported to the user through the plugin itself.

The other possibility provided by the tool is to use testing in order to verify refinement checking. In this case, the tool automatically performs test case generation (more on this is presented in Section 4) from the behavioral model of the more abstract requirement and then executes the generated tests on the behavioral model of the more concrete requirement.

The screenshot in Fig. 11 shows how the GUI of the plugin appears. In this screenshot it is possible to see the panel where hierarchical requirements and their relationships are graphically modeled and the view of the Verification plugin in the lower pane with the Refinement Checking tab where the plugin shows the extracted derive relationships. In this tab, the user can make selections and start refinement checking verification. In the example shown in Fig. 11, about an airbag system use case, it is possible to see two requirements bound by the DeriveReqt relationship. The two requirements shown in the example have behavioral models, expressed as state machines attached to them. Each state machine is specified in a separate diagram, using the same modeling environment.

Fig. 9 shows the state machine diagram of the more abstract requirement (Deploy_should_be_preceded_by_critical_collision), which expresses a precedence relationship: the deploy event must always be preceded by the critical_ collision event. The more concrete requirement (Deploy_ within_two_time_ticks_after_collision) has a state machine shown in Fig. 8.

In order to verify that the refinement relationship holds between the two requirements (i.e. that Deploy_within_two_time_ticks_after_collision correctly refines Deploy_should_be_preceded_by_critical_collision) we have to select the Deploy_should_be_preceded_by_critical_collision abstract requirement in the list of relationships, as shown in Fig. 11, and push the Start verification button (the Refresh button is used to create the list of relationships or to update it after modifications to the model).

Fig. 7 shows the verification report, indicating that the refinement relationship does not hold. In order to understand why refinement does not hold it is possible to look at the counterexample (init -> [start] -> [collision] -> deploy) which is a sequence of events that may occur in the concrete behavior but not in the abstract one, thus invalidating refinement. Events enclosed in square brackets in the counterexample represent hidden events, i.e. events that are not considered because they are not shared by the two behavioral models. In this particular case, it is possible to see that deploy can occur without being preceded by critical_collision (it is preceded by collision, which is a hidden event). The problem arises because in the descriptions of the two requirements two different events have been used to represent a collision (collision in one case, critical_collision in the other case). The problem can be fixed by using the same collision event in the two requirements or by specifying that these two events should be considered the same in the verification of refinement. This can be done by selecting the refinement relationship and by adding the mapping of the two events. After this fix, if we repeat refinement checking, we get the result that the relationship holds (valid).

The CSP verification model generated by the plugin for this verification task is shown in Listing 1. In it, the Implementation process corresponds to the process indicated as I in Section 3.1 while the first two defined processes are the ones automatically generated from the two state machines.

4. Automated generation of requirements-based test cases

Testing is the main verification technique recommended by ISO 26262. Fig. 10 illustrates the process for testing a system under test (SUT) against a set of requirements (requirements-based testing):

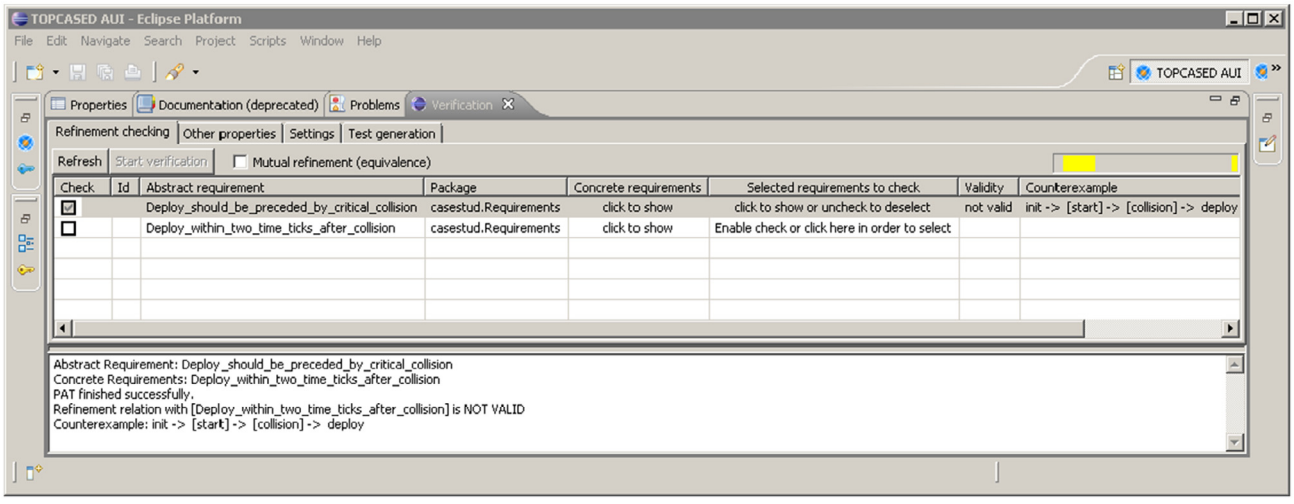


Fig. 7. The verification report with a failed verification (and counter example).

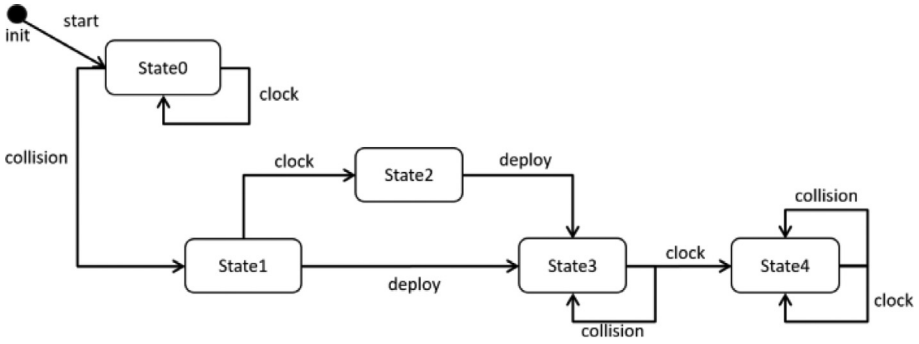


Fig. 9. The state machine attached to the more abstract requirement.

```
//============Process Definitions==================================
stm_Deploy_should_be_preceded_by_critical_collision()=((init()));
init()=idle();
ownedBehav_stm_Deploy_should_be_preceded_by_critical_collision()=stm_Deploy_should_be_preceded_by_critical_collision();
St3()=((no_deploy->St3())[]finalState0());
St2()=((deploy->St3())[](no_deploy->St2()));
finalState0()=Skip;
idle()=(critical_collision->St2());
//====================Process Definitions==========================
Deploy_within_two_time_ticks_after_collision_init_deploy_within_two_time_ticks()=(start-
>Deploy_within_two_time_ticks_after_collision_State0());
stm_Deploy_within_two_time_ticks_after_collision()=((Deploy_within_two_time_ticks_after_collision_init_deploy_within_two_time_ticks()));
Deploy_within_two_time_ticks_after_collision_State4()= ((collision->Deploy_within_two_time_ticks_after_collision_State4())[](clock-
>Deploy_within_two_time_ticks_after_collision_State4()));
Deploy_within_two_time_ticks_after_collision_State1()= ((clock->Deploy_within_two_time_ticks_after_collision_State2())[](deploy-
>Deploy_within_two_time_ticks_after_collision_State3()));
Deploy_within_two_time_ticks_after_collision_State0()= ((collision->Deploy_within_two_time_ticks_after_collision_State1())[](clock-
>Deploy_within_two_time_ticks_after_collision_State0()));
Deploy_within_two_time_ticks_after_collision_State3()= ((clock->Deploy_within_two_time_ticks_after_collision_State4())[](collision-
>Deploy_within_two_time_ticks_after_collision_State3()));
Deploy_within_two_time_ticks_after_collision_State2()= (deploy->Deploy_within_two_time_ticks_after_collision_State3());
PreImplementation_Deploy_within_two_time_ticks_after_collision() = (stm_Deploy_within_two_time_ticks_after_collision());
Implementation() = (PreImplementation_Deploy_within_two_time_ticks_after_collision())\{collision, start, clock};
#assert Implementation() refines stm_Deploy_should_be_preceded_by_critical_collision();
```

Listing 1. The CSP model for the sample refinement checking.

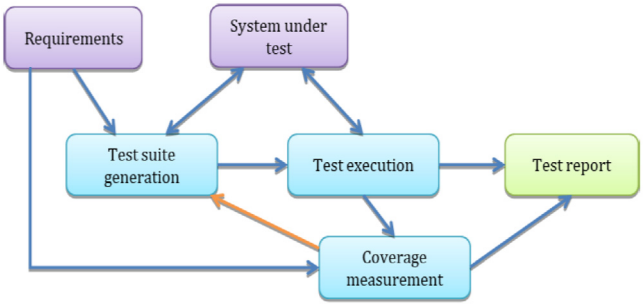


Fig. 10. Requirements-based testing.

first, a test suite, i.e. a set of test cases covering all the safety requirements, is created. A test case consists of input stimuli and the expected outcome. Test execution runs the SUT feeding it with the inputs specified by each test case and compares the SUT output against the expected one. Coverage measurement assesses whether the test suite is sufficiently complete; if not, further test cases have to be created so as to reach the target coverage. As an alternative to manual test case generation, Model-Based Testing (MBT) consists of deriving test cases from an abstract model (for requirements-based testing, a model of the requirements has to be used). This section shows how the formal behavioral models derived from SysML models (and associated to safety requirements) are used in our approach for automatic generation of test cases that can be used for requirements-based testing and for verifying software safety requirements according to ISO 26262. The size of the set of generated test cases can be subsequently optimized by means of simulation-based algorithms that prune redundant test cases. Note that requirements-based testing is just one of the several test methods recommended by ISO-26262. The use of a variety of test methods contributes to greater diversity and better opportunity to catch errors.

Most of the existing literature about MBT of reactive systems starting from formal models is based on Finite State Machine (FSM) and Labeled Transition System (LTS) models [12]. As these models are very general, many other specification formalisms (e.g. process algebras, petri nets, extended/hierarchical state machine models), including the ones supported by SysML, can be interpreted according to these two basic models. This opens the possibility to apply the same MBT techniques to a variety of modeling languages. As already mentioned, even if we consider UML and UML-based specification languages, which are semi-formal, it is possible to derive corresponding formal models from some parts of the UML models, provided the semantic variation points in the UML models are resolved in some way. For example, using this approach, it is possible to give a formal semantics to UML state machines, which enables their interpretation in terms of LTSs.

One limitation of FSM based methods is that they typically consider only deterministic machines. Of course, nondeterministic machines can be turned into deterministic machines, but at the cost of state explosion. This feature makes FSM-based methods inadequate for large nondeterministic models. One good point of

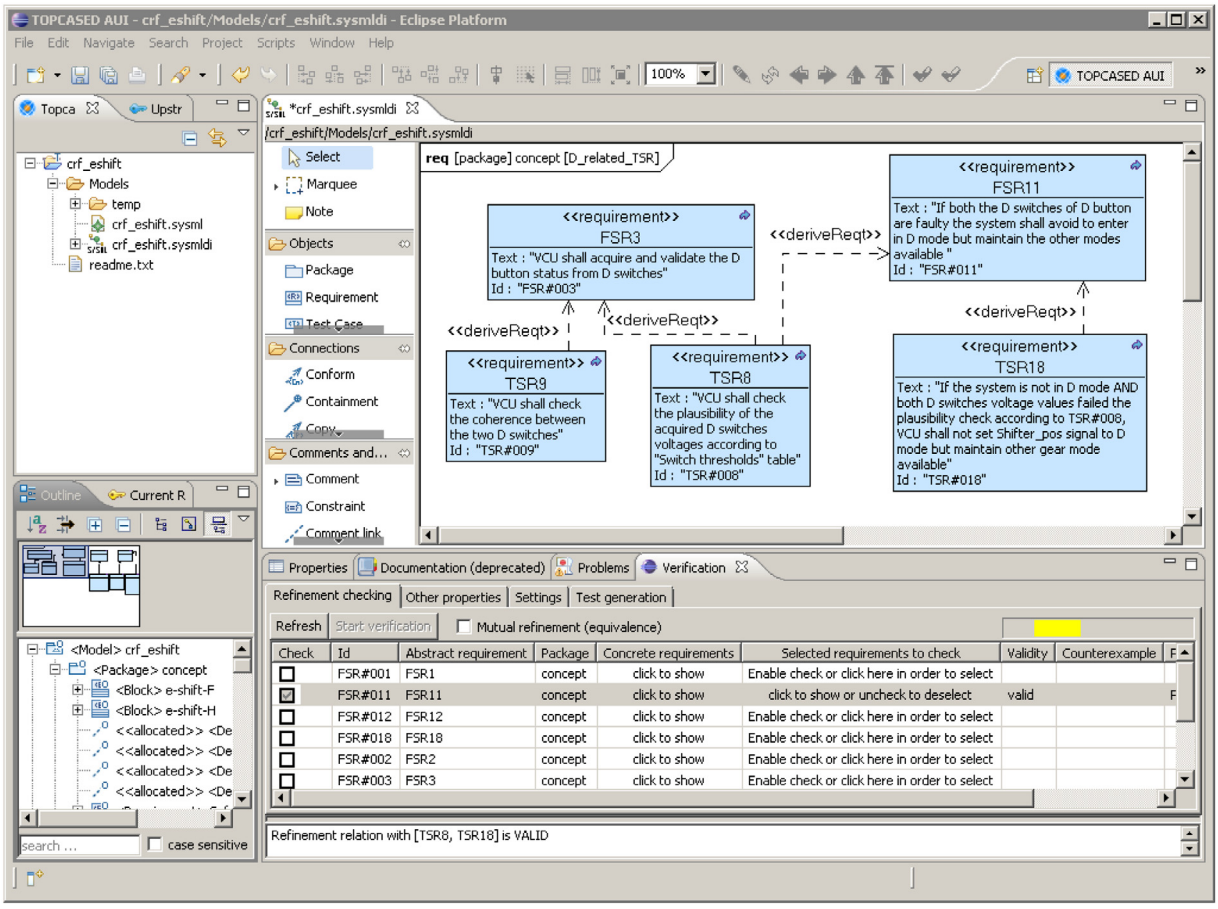


Fig. 11. The GUI of the plugin.

FSM-based methods, however, is that they have been highly optimized with the search for minimal tests [13]. LTS methods are more general in that they deal with nondeterministic and infinite-state systems. With large or infinite-state systems, lazy (on-the-fly) evaluation is typically used. Because of their ability to deal with nondeterministic and infinite-state systems, LTS-based methods seem more appropriate for our context. In particular, one of the LTS-based theories that seems most appropriate is the one that deals with I/O LTSs (chapter 7 of [12]). This theory includes the definition of several implementation relationships and methods to build test suites.

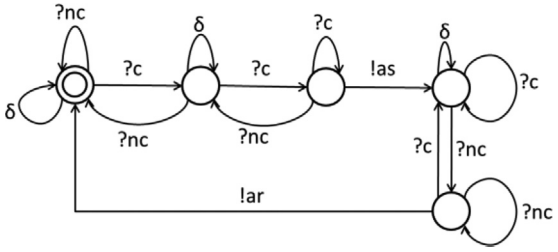


Fig. 12. An example of I/O LTS.

4.1. MBT with I/O LTSs

According to the theory of MBT with I/O LTSs, the behavioral model of the SUT associated with a safety requirement (i.e. the specification) is converted into an I/O LTS (an LTS with labels divided into input, output and internal). The model must be such that it does never refuse inputs (all inputs are available in all states; this can be obtained by completing an incomplete specification by saying, for example, that non-specified inputs are ignored, i.e. by adding self-loops for non-specified inputs in all states). If the model has quiescent states (i.e. states where no outputs can occur) these are marked by adding a self-loop to that state, labeled with a special δ label (this could be interpreted as the expiration of a timeout in case the environment is waiting for an output). Fig. 12 shows an example of an I/O LTS that does never refuse inputs, with two inputs (?c and ?nc) and two outputs (!as and !ar), and the additional δ events already added.

Test generation can target one or many different implementation relationships. The simplest relationship is traces refinement (see Section 3). The most used relationship is ioco (input-output conformance) [14]. This relationship is defined considering the traces of the specification (with the inclusion of the special event δ). If, after one of these traces, the outputs that the implementation can perform are a subset of the outputs that the specification can perform after the same trace, the relationship holds. This relationship is weaker than traces refinement (traces refinement implies ioco) because ioco does not constrain the implementation in any way after the traces that the specification cannot execute. However, if both the implementation and the specification have the same alphabet of events, do never refuse inputs, and are completed with events, then the two relationships coincide. Each test case generated from a model is again an I/O LTS with final states that are labeled as success, fail (and inconclusive). This LTS has inputs and outputs reversed w.r.t. the specification. In each state it can

either emit exactly one output (i.e. an input to the system) or receive any input (i.e. any output from the system) or the special event that corresponds to δ (the timeout that indicates quiescence of the system). Applying a test case means executing it step by step until a final state is reached. The label of the final state determines the result of the test case. If the system is nondeterministic, each test case should be repeated several times and we can say that the test case has been passed only if all runs end up with the pass result.

4.2. Toolchain for MBT

In our approach, MBT is supported by performing test case generation from the behavioral models associated with safety requirements. These models are automatically completed with inputs (so that they can never refuse to perform inputs) and with δ events. When using MBT for testing refinement, the SUT model is also completed in the same way, and events that occur only in the implementation are considered as internal (i.e. nor input neither output). In this way, we make sure ioco and traces refinement coincide.

The toolchain for performing MBT in our demonstrative Topcased plugin is based on CADP/TGV [15], an already existing tool which can generate test cases for testing the ioco relationship on I/O LTS models automatically. The output of TGV is a collection of test cases, each one represented by an I/O LTS with three sets of trap states: Pass, Fail and Inconclusive, that represent the verdict. The input of TGV consists of two I/O LTSs: a specification and a test purpose. The specification is a model of the expected behavior of the SUT. A test purpose is the representation of the part of behavior of the SUT that must be taken into consideration for test generation. This can be used, for example, in order to limit the size of each resulting test case. A test purpose must be complete, i.e. it must be such that all events must be possible in all states, and it must include two sets of special states called trap states: the set of accept states and the set of refuse states. Each accept or refuse state must include self-loops for all the events (this is why it is called a trap state). The meaning of an accept state is that, when it is reached, the behavior we are interested in has been observed. Instead, a refuse state is one that, when reached, means the observed behavior is not relevant for the test. In both cases the test must be stopped. If an accept state is reached and all outputs from the SUT have been among the expected ones (as specified in the specification), an accept verdict is generated. Otherwise, if a refuse state is reached and all outputs from the SUT have been among the expected ones, an inconclusive verdict is emitted. Finally, a fail verdict is generated if the SUT generates unexpected outputs, but this remains implicit in the output I/O LTS (if the output is not present, this means fail). TGV includes the possibility to complete incomplete test purposes, by adding self-loops for all the missing events. Moreover, TGV automatically adds δ events as necessary. The generation of δ events is performed by TGV when it combines the specification and the test purpose into a single I/O LTS. For test case generation, TGV uses a lazy evaluation technique that lets it limit the amount of memory consumed. TGV can take inputs specified in several input languages and convert them into the I/O LTS form. Unfortunately, TGV cannot take CSP as input. However, it can take LOTOS[2] [16], which is another formal specification language similar to CSP. As LOTOS was inspired by CSP, its process algebra is very similar to CSP. Note that TGV was selected as the best fit to our problem, as we didn't find

[2] LOTOS is a specification language that has been specifically developed for the formal description of the OSI (Open Systems Interconnection) architecture, although it is applicable to distributed, concurrent systems in general.

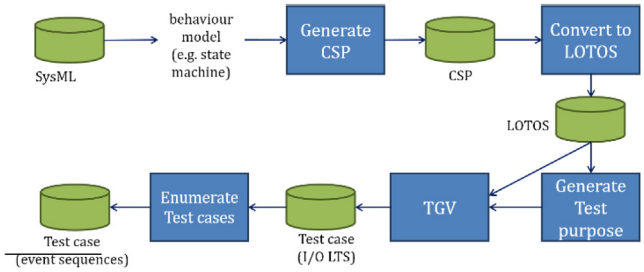


Fig. 13. The toolchain for performing MBT.

MBT tools capable of performing test case generation based on the ioco relationship that can take directly CSP as input.

As we already have CSP models generated from SysML behavioral models, we created a conversion tool from CSP to LOTOS, in order to enable the use of CADP TGV for MBT test case generation. As in LOTOS the distinction between input and output events may not be possible, a separate file is used by TGV in order to give this specification (i.e. whether each event has to be considered as input or output). This is also generated automatically by the converter. Finally, in order to limit the manual work of the user as much as possible, an automatic test purpose generator has been created. Test purposes can then be edited by the user if necessary. The complete MBT toolchain resulting from this approach is shown in Fig. 13.

This basic method has been integrated into our plugin for the Topcased [17] environment along with the refinement checking tool. Having these two tools in the same plugin makes code reuse straightforward and provides a single environment where users can develop software safety requirement models in SysML, check refinement relationships among requirements using the refinement checking tool and generate test cases starting from the safety requirements models.

5. Experimental results

An evaluation of the approach has been done by means of the demonstrative toolchain on a use case proposed by CRF (Centro Ricerche Fiat), the FCA (Fiat Chrysler Automobiles) Research Center, within the VeTeSS (Verification and Testing to Support Functional Safety Standards) European Project (https://artemis-ia.eu/project/43-vetess.html).

VeTeSS developed standardized tools and methods to verify the robustness of safety-relevant systems, particularly against transient common-cause faults. CRF's use case selected for the project was an electric gear selector for HEV/EV (e-shift [18]): the system provides mechanical locking or unlocking of the transmission when the parking mode is selected (by the driver or automatically), avoiding unwanted movement of the vehicle when stopped.

The electric gear selector is composed of a Vehicle Control Unit (implementing the control strategies), a number of switches (sensing of Parking, Drive, Rear and Neutral buttons) and a parking lock actuator (actuating the command to a park pawl motor). Two of the identified safety goals were analyzed in the project: avoiding an unwanted unlocking of the parking brake (SG#1) and avoiding an unwanted activation of the parking brake when the car is moving (SG#2). Starting from an already existing model of the e-shift system and a natural language expression of its safety requirements, a corresponding SysML model has been created, including the formalization of safety requirements according to the proposed approach. This work has been done by a team of CRF's engineers having background in SysML but not in formal methods, with the support of the tool developers from the Turin Polytechnic.

Unfortunately, for the time being, software safety requirements are not available for the e-shift use case. Only system-level requirements are available (functional and technical safety requirements). For this reason, the evaluation has been done using the available requirements. This means that compliance and consistency have been checked according to our approach between functional safety requirements and technical safety requirements. Similarly, test cases have been generated from technical safety requirements rather than from software safety requirements. However, we think this kind of experiment is still significant for demonstrating and evaluating the approach, and it is also relevant for software, considering that the test cases derived from the technical safety requirements can be used for software-in-the-loop testing. Even if not all the available safety requirements have been formalized, a significant subset of them has been formalized in SysML, with their associated behavior. For these requirements, the plugin has been able to check refinement relationships, and for technical safety requirements it has been able to generate test cases, as expected. The subset of formalized requirements includes safety goal SG#2, 9 functional safety requirements, and the 18 technical safety requirements derived from them. These include all the requirements related to the D button. For one of the requirements (TSR#008) the test cases generated by the tool have been manually translated into Simulink-related time histories. As an example, the test case generation for one of the technical safety requirements (TSR#011) is presented here. The text of the requirement is: "If the system is in P mode, VCU shall set Shifter_pos signal to D mode only if brake pedal status is pressed and validated". This requirement has been formalized using the state machine shown in Fig. 14.

The state machine consists of a main state (State 1), composed of two regions (representing parallel behaviors). The lower region represents the evolution of the brake pedal status, while the upper region represents the evolution of the driving mode. The meaning of the events occurring in the state machine is defined in Table 1. The lower state machine models the brake pedal status by means of two states: PressedAndValidated and Other. Initially the state is Other, but it can change at any time. The state machine expresses the fact that when the brake pedal status is in the "PressedAndValidated" state the result of the acquisition/validation of the brake pedal status is valid-pressed (event B_valid_pressed) while when the state is "Other", the result can be valid-unpressed or invalid. The upper state machine models the driving mode by means of two states: PMode and OtherMode. Inside PMode there are two sub-states: PState1 and PState2.

Table 1
Events meaning for requirement #011.

P_in	The P driving mode is entered
D_in	The D driving mode is entered
R_in	The R driving mode is entered
N_in	The N driving mode is entered
B_valid_pressed	the result of the acquisition/validation of the brake pedal status is communicated and this result is valid-pressed
B_valid_unpressed	the result of the acquisition/validation of the brake pedal status is communicated and this result is valid-unpressed
B_invalid	the result of the acquisition/validation of the brake pedal status is communicated and this result is invalid
shifter_pos_to_D	the Shifter_pos signal is set to D mode

The latter state is reached only after event B_valid_pressed, i.e. only when the result of the last acquisition/validation of the brake pedal status has been valid-pressed. As evident from the diagram, this machine expresses the fact that the Shifter_pos signal can be set to D mode (i.e. event shift_pos_to_D can occur) only in PState2, i.e. only when the result of the last acquisition/validation of the brake pedal status has been valid-pressed. When considering this requirement alone, the output that has to be observed is event shift_pos_to_D, while all the other events can be considered as inputs. Accordingly, when deriving test cases for this requirement, shift_pos_to_D is classified as output while all the other events are classified as inputs. The generation of the test cases with a default test purpose set with a maximum length of the test patterns equal to 100 produces a test case I/O LTS composed of a total of 26,900 steps (i.e., transitions of the I/O LTS). The time taken by TGV to generate the test cases (with the default test purpose set to a maximum test length of 100) is in the order of few seconds. Considering that a maximum test length of 100 is already a fairly high value, this result gives a positive assessment about the scalability of the tool used for test generation. This experimental work

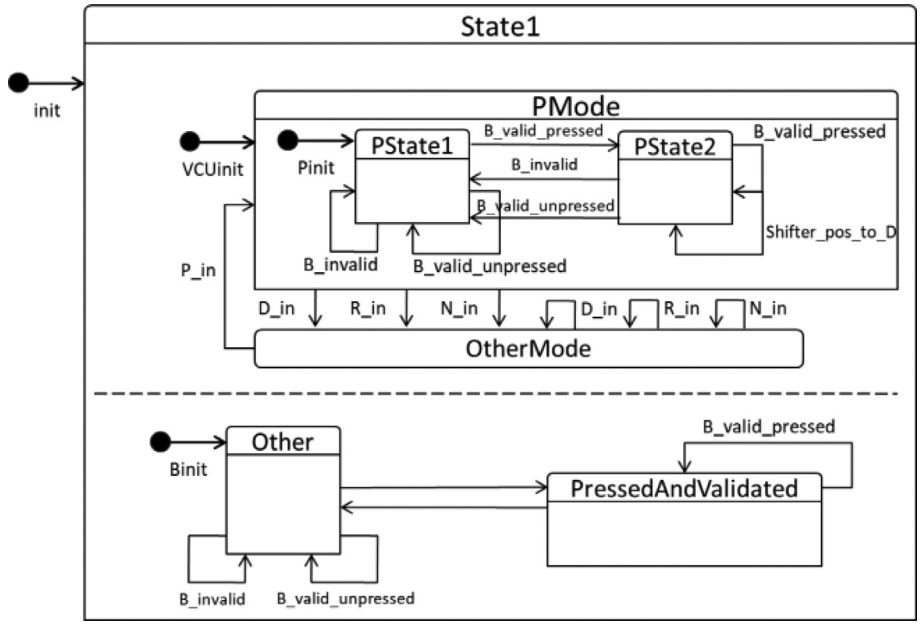


Fig. 14. The formalization of TSR#011 from the e-shift use case.

allowed us to get a confirmation that refinement checking can be applied with success to a realistic industrial use case. As requirements behavioral models are not so complex, the time taken by refinement checking is absolutely affordable (each verification takes seconds to complete). In the experimentation with the e-shift use case, refinement checking proved to be a tool useful for obtaining coherent and precise requirements models. In fact, modeling errors were spotted either by performing simulation (made possible by the integration with PAT) or by performing refinement checking. The final models were obtained after some check-fix iterations. The main issues that had to be corrected during these iterations were related to wrong uses of the modeling language. The modeling work also led to the detection of a redundancy in the set of functional safety requirements: two of the original requirements were found to be equivalent, i.e. to have the same behavioral model. The textual formulation of the e-shift requirements was not precise enough for their formalization, and some of the implicit underlying assumptions made by requirements engineers had to be understood and made explicit in order to proceed with formalization. The formalization work necessary as a pre-condition for performing verification by means of refinement checking gave the opportunity to improve the precision of safety requirements, while refinement checking provided the confidence that derivations were sound or spotted conceptual errors or lack of necessary assumptions in requirements formalization. One of the main limitations observed by CRF's engineers is related to the difficulty of formalizing requirements as state machines or as one of the other behavioral models available in SysML. However, CRF agreed, in their final evaluation report, that although the effort for the formalization of requirements as state machines is considerable, it was feasible and it made formal verification possible, thus improving the quality of the process. Without this approach and tool support, formal verification would not have been possible, because of the lack of specific skills in formal methods. The benefits of having performed this modeling and formal verification step that the CRF team reported as most useful are the resulting non-ambiguity, consistency, and coherence of the obtained requirements set: safety requirements have precise meaning, and each one derives correctly from the higher-level ones. By employing the proposed formal verification tool, this result could be achieved with acceptable effort. Another result is that the team of engineers was able to complete all the formal specification and verification tasks of this project with limited support and guidance from the tool developers, which confirms that the proposed toolchain can be used by engineers with background in SysML. The main thing that they learned from the process is that a formal specification and verification step can really help to improve the quality of requirements.

6. Related work

After the publication of ISO 26262 some proposals have appeared in the literature about how to introduce formal methods in ISO 26262 compliant development processes. However, only in some cases these works address the problem of making the use of formal methods user-friendly and fully automated, by hiding the complexity of formal notation and reasoning to the final user.

For example, [19] presents a methodology that includes formal verification of refinement relationships between safety requirements and their software implementations. Differently from our work, it requires that requirements are expressed directly in the formal language of the verification tool, and the tool (a theorem prover) is not fully automatic. Another work [20] provides support for the verification of safety requirements by exploiting syntactic contract conditions. The approach proposed in [20] as a way to limit the exposure of the user to formal languages consists of avoiding a full formalization of requirements. In this way, however, formal verification remain possible only partially, while our approach allows a full formalization and verification of requirements. Instead, [21] proposes the use of CSP-based automated verification tools in order to formally check safety requirements. While the paper envisages the usefulness of translating safety requirements into CSP automatically, this is left as future work. [22] presents a tool called SESAMM which hides the complexity of formal specifications by exploiting specification patterns. The user specifies requirements in terms of patterns and the latter are then translated automatically into formal notation. While the SESAMM approach is a possible valid alternative to our approach to create formal requirements specifications in a user-friendly way, currently SESAMM does not address the problem of refinement checking as we do, i.e. it cannot check automatically that a derived requirement correctly implements a more abstract one.

Among the works that use formal methods and address user-friendliness and automation in a way more similar to what we propose, [23] presents a methodology based on expressing requirements and system models in UML and translating them automatically into formal notations that can then be used for verification by model checking. Although there are similarities between our approach and the one proposed by [23], e.g. the extraction of formal models from UML behavioral diagrams, there are also substantial differences: we consider SysML rather than UML, which allows us to represent requirements derivations, and we use refinement checking rather than model checking. In this way, we can compare requirements at different abstraction levels, and we avoid to expose the user to a specialized formalism for requirements (in [23], requirements have to be specified by means of pre and post conditions). In addition, we introduce the possibility of a smooth transition from refinement checking to testing, via automatic generation of requirements-based test cases.

Another work related to our one is [18], where the same e-shift use case has been used for evaluation. However, this paper proposes a technique for improving test coverage, which is complementary to our approach rather than alternative to it.

In the rest of this section, we give an overview of related work in the area of refinement checking. Theories of refinement have been developed for several formal specification languages (e.g. [24–26]). Among these languages, CSP [7,8] has gained particular interest and popularity, also because of the availability of widespread techniques and tools for automated checking of refinement in CSP [27,10]. The CSP theory of refinement has been used in several fields (software, hardware and system development) with the aim of verifying the compliance of implementations with specifications or the compliance of specifications with other corresponding more abstract specifications. However, as CSP is typically outside the common background of engineers, some attempts have been made in order to hide CSP to final users. This can be achieved by translating the models used for specification into CSP. For example, [28] presents an approach for using CSP-based refinement checking for verifying mobile phone applications: properties are expressed as constrained English sentences that can be automatically translated into CSP and another CSP model is automatically built from the code of the application. Then refinement checking is used in order to verify that the code fulfills the properties. Another example is [29] where OCCAM programs are automatically translated into CSP in order to apply refinement checking. Our approach is similar to these ones, but the context (ISO 26262 safety requirements) is totally different.

Focusing on UML-based specification formalisms, some attempts have been made to facilitate refinement checking of UML models, by automatically deriving CSP processes from (parts of) UML models. For example, [30] presents a CSP semantics for a subset of UML that includes UML state machines while [31,32] deal with automatic translation of UML state machines and UML

activity diagrams respectively into CSP. Given the rich literature in this field, in our work we do not provide alternative ways for generating CSP from UML behavioral diagrams but we exploit the already existing approaches.

7. Conclusions

Starting from the observation that electronic and programmable automotive systems are becoming more and more complex while playing an increasingly crucial role, we proposed an automated, approach to formal modeling and verification of requirements according to the recently issued standard ISO 26262. We acknowledge that formal verification techniques are often seen as complex and highly sophisticated methods that may not be easily applicable to concrete industrial use cases. In this work we have addressed exactly this challenge by only minimally exposing the theoretical complexity of the approach. In fact, system models and behaviors are expressed by means of SysML, a well-known modelling language that most people with an industrial background are already familiar with. According to our approach, demonstrated as a plugin implementation in the Topcased environment, formal models are extracted from the SysML requirements specifications, by resolving variation points. These models can then be used to formally verify, in an automated way, that requirements, with their corresponding behavioral diagrams, expressed at different levels of abstraction, are related by the correct relationships specified at system design time. Moreover, starting from the same formal models, it is possible to automatically generate test cases that can be used for requirements-based testing, an alternative way to verify refinement relationships.

The fact that the proposed methodology hides the complexity of formal notations to the user is an important result about usability. In order to also prove the validity of our solution in real world use cases, we applied the methodology and its demonstrative tool chain to one main example derived from industrial use cases, which is an e-shift use case proposed by FCA. According to the results of the experiment, in which a team of engineers from CRF without specific training in formal methods performed safety requirements specification and verification, the proposed approach was evaluated feasible and user friendly enough by the engineers, also showing adequate performance of the automated tools employed. At the same time, it was recognized by the engineers involved that the formalization of requirements, even with the proposed approach based on state machines, requires considerable effort. However, this effort permitted the achievement of a formal verification of requirements which would not be possible to obtain otherwise without employing personnel with specific training in formal methods.

We leave as future work a more extensive evaluation of the proposed approach and toolchain on other use cases and with other teams of engineers.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The work described in this paper has been partially supported by the VeTeSS European research project (grant agreements no. 295311).

References

[1] ISO, ISO 26262:2018, Road vehicles – Functional safety, 2018.
[2] M. Nyberg, D. Gurov, C. Lidström, A. Rasmusson, J. Westman, Formal verification in automotive industry: enablers and obstacles. In: T. Margaria, B. Steffen (Eds.), Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice. ISoLA 2018. Lecture Notes in Computer Science, vol 11247. Springer, Cham, 2018.
[3] SysML.org, Sysml open source specification project, 2015. [Online]. Available: http://sysml.org/.
[4] Sanford Friedenthal, Alan Moore, Rick Steiner, A Practical Guide to SysML: The Systems Modeling Language, Morgan Kaufmann, 2014.
[5] S. Wolny, A. Mazak, C. Carpella, et al., Thirteen years of SysML: a systematic mapping study, Softw. Syst. Model. (2019), https://doi.org/10.1007/s10270-019-00735-y.
[6] EAST-ADL Domain Model Specification, Version V2.1.12, EAST-ADL Association, 2013. [Online] Available: https://www.east-adl.info/Specification/V2.1.12/EAST-ADL-Specification_V2.1.12.pdf.
[7] R. Alur, R. Grosu, and B.-Y. Wang, "Automated refinement checking for asynchronous processes," in Formal Methods in Computer-Aided Design, ser. Lecture Notes in Computer Science, J. Hunt, WarrenA. and S. Johnson, Eds. Springer Berlin Heidelberg, 2000, vol. 1954, pp. 74–91. [Online]. Available: http://dx.doi.org/10.1007/3-540-40922-X_5.
[8] A.W. Roscoe, C.A.R. Hoare, R. Bird, The Theory and Practice of Concurrency, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
[9] S. Liu, Y. Liu, t. André, C. Choppy, J. Sun, B. Wadhwa, J. Dong, A formal semantics for complete uml state machines with communications, in: Integrated Formal Methods, ser. Lecture Notes in Computer Science, E. Johnsen and L. Petre, Eds. Springer Berlin Heidelberg, 2013, vol. 7940, pp. 331–346. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38613-8_23.
[10] J. Sun, Y. Liu, J. Dong, Model checking csp revisited: Introducing a process analysis toolkit, in: Leveraging Applications of Formal Methods, Verification and Validation, ser. Communications in Computer and Information Science, T. Margaria and B. Steffen, Eds. Springer Berlin Heidelberg, 2008, vol. 17, pp. 307–322. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-88479-8_22.
[11] D. Makartetskiy, R. Sisto, An approach to refinement checking of sysml requirements, in: Emerging Technologies Factory Automation (ETFA), 2011 IEEE 16th Conference on, Sept 2011, pp. 1–4.
[12] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, A. Pretschner, Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science), Springer-Verlag, New York Inc., Secaucus, NJ, USA, 2005.
[13] Q. Guo, R. Hierons, M. Harman, K. Derderian, Computing unique input/output sequences using genetic algorithms, in: Formal Approaches to Software Testing, ser. Lecture Notes in Computer Science, A. Petrenko and A. Ulrich, Eds. Springer Berlin Heidelberg, 2004, vol. 2931, pp. 164–177. [Online]. Available: http://dx.doi.org/10. 1007/978-3-540-24617-6_12.
[14] Jan Tretmans, Test generation with inputs, outputs and repetitive quiescence, Software-Concepts and Tools 17 (3) (1996) 103–120.
[15] H. Garavel, F. Lang, R. Mateescu, W. Serwe, Cadp 2011: a toolbox for the construction and analysis of distributed processes, Int. J. Software Tools Technol. Transfer 15 (2) (2013) 89–107 [Online]. Available: http: //dx.doi.org/10.1007/s10009-012-0244-z.
[16] Bolognesi, Tommaso, Ed Brinksma. Introduction to the ISO specification language LOTOS. Computer Networks and ISDN systems 14.1 (1987) 25–59.
[17] M. Pantel et al., The topcased project – a toolkit in open source for critical applications and systems design.
[18] A.B. Hocking, J. Knight, M.A. Aiello, S. Shiraishi, Arguing software compliance with iso 26262, in: 2014 IEEE international symposium on software.
[19] A. Nellis, P. Kesseli, P.R. Conmy, D. Kroening, P. Schrammel, M. Tautschnig, Assisted coverage closure, in: NASA formal methods symposium, Springer, Cham, 2016, pp. 49–64.
[20] Jonas Westman, Mattias Nyberg, Providing tool support for specifying safety-critical systems by enforcing syntactic contract conditions, Requirements Eng. (2018) 1–26.
[21] Reliability Engineering Workshops, Nov 2014, pp. 226–231.
[22] P. Filipovikj, T. Jagerfield, M. Nyberg, G. Rodriguez-Navas, C. Seceleanu, Integrating pattern-based formal requirements specification in an industrial tool-chain. In 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC) (Vol. 2, pp. 167–173). IEEE, 2016, June.
[23] N.J. Tudor, J. Botham, Proving properties of automotive systems of systems under iso 26262 using automated formal methods, in: 9th IET International Conference on System Safety and Cyber Security (2014), Oct 2014, pp. 1–6.
[24] G. Bahig, A. El-Kadi, Formal verification of automotive design in compliance with iso 26262 design verification guidelines, IEEE Access 5 (2017) 4505–4516.
[25] J. Derrick, E. Boiten, Refinement in Z and Object-Z: Foundations and Advanced Applications, Springer-Verlag, London, UK, UK, 2001.
[26] J.-R. Abrial, Modeling in Event-B: System and Software Engineering, 1st ed., Cambridge University Press, New York, NY, USA, 2010.
[27] C.A.R. Hoare, Communicating sequential processes, Commun. ACM, vol. 21, no. 8, pp. 666–677, Aug. 1978. [Online]. Available: http://doi.acm.org/10.1145/359576.359585.
[28] S. Kundu, S. Lerner, R. Gupta, Automated refinement checking of concurrent systems, in: Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM International Conference on, Nov 2007, pp. 318–325.

[29] B. Aichernig, E. Jöbstl, M. Kegele, Incremental refinement checking for test case generation, in: Tests and Proofs, ser. Lecture Notes in Computer Science, M. Veanes and L. Viganó, Eds. Springer Berlin Heidelberg, 2013, vol. 7942, pp. 1–19. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38916-0_1.
[30] F.R. Barnes, C.G. Ritson, Checking process-oriented operating system behaviour using csp and refinement, SIGOPS Oper. Syst. Rev., vol. 43, no. 4, pp. 45–49, Jan. 2010. [Online]. Available: http://doi.acm.org/10.1145/1713254.1713265.
[31] J. Davies, C. Crichton, Concurrency and refinement in the unified modeling language, Formal Aspects of Computing 15 (2-3) (2003) 118–145, https://doi.org/10.1007/s00165-003-0008-3.
[32] D. Varró, M. Asztalos, D. Bisztray, A. Boronat, D.-H. Dang, R. Geiß, J. Greenyer, P. Van Gorp, O. Kniemeyer, A. Narayanan, E. Rencis, E. Weinell, Applications of graph transformations with industrial relevance, In: A. Schürr, M. Nagl, A. Zündorf, (Eds.), Berlin, Heidelberg: Springer-Verlag, 2008, ch. Transformation of UML Models to CSP: A Case Study for Graph Transformation Tools, pp. 540–565. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-89020-1_36.

Denis Makartetskiy received the M.S. degree in computer science from the Ufa State Aviation Technical University (Ufa, Russian Federation) in 2009. He was a research assistant at Polytechnic University of Turin from 2010 to 2014 working on application of verification techniques for automotive industry in context of ISO 26262. S Since 2014 he is a Process Consultant and Leader of Expert Area "Functional Safety" at Kugler Maag CIE GbmH. He supports automotive semiconductor companies, Tier 1 suppliers, OEMs and software companies in achieving compliance with ISO 26262 by trainings, consulting services, safety audits and assessments.

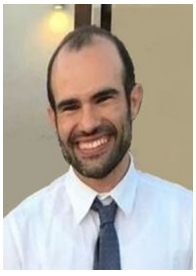
Guido Marchetto is an assistant professor at the Department of Control and Computer Engineering of Politecnico di Torino. He got his Ph.D. in Computer Engineering in April 2008 from Politecnico di Torino. His research topics cover distributed systems and formal verification of systems and protocols. His interests also include network protocols and network architectures.

Riccardo Sisto received the Ph.D. degree in Computer Engineering in 1992, from Politecnico di Torino, Italy. Since 2004, he is Full Professor of Computer Engineering at Politecnico di Torino. His main research interests are in the area of formal methods, applied to distributed software and communication protocol engineering, distributed systems, and computer security. He has authored and co-authored more than 100 scientific papers. He is a Senior Member of the ACM.

Fulvio Valenza received the M.Sc. (summa cum laude) in 2013 and the Ph.D. (summa cum laude) in Computer Engineering in 2017 from the Politecnico di Torino, Torino, Italy. His research activity focus on network security policies. Currently he is a Researcher at the Politecnico Torino, Italy, where he works on orchestration and management of network security functions in the context of SDN/NFV-based networks.

Matteo Virgilio received the M.S. degree in Computer Engineering from Politecnico di Torino, Italy, in 2012. Currently, He is a Ph.D. student in Control and Computer Engineering at Politecnico di Torino. His research interests include innovative network protocols and architectures, Content Centric Networking and formal verification techniques applied in the context SDN/NFV.

Denise Leri graduated in Nuclear and Subnuclear Physics in 2011, from Università degli Studi di Torino, Torino, Italy. Since 2014 she has been working at CRF - Centro Ricerche FIAT, first as a external consultant, then as Functional Safety Specialist on ADAS Systems. Since the beginning of her working activity, her main interests have been in the area of formal methods and tools, applied to Functional Safety, SOTIF analysis and ADAS system implementation. Denise Leri is currently a FCA employee.



Paolo Denti graduated in Electronic Engineering in 2010 from Università degli Studi di Cagliari, Italy. From 2010 to 2012 he had worked on several International Biometrics projects during the university career. From end of 2012 to 2013 he had an advanced training course in the automotive field, including an internship in the ëSoftware Factory and Methodologiesí group at CRF. Since then, has been working at Centro Ricerche FIAT as E/E Architectures &SW Factory Specialist. Since the beginning at CRF he has been involved in several projects focused on next-generation electrified powertrain, driver assistance and telematics functionalities.

Roberto Finizio graduated in Physics in 1999 from Univerisità degli Studi di Torino, Italy. Since 2000 he has been working for Centro Ricerche FIAT, first as a researcher in Micro & Nano Technologies Department and since 2008 as senior specialist in Advanced Electrical/Electronics Department. Since 2012 he was appointed as Manager of Electronic Architecture and Integration Unit active in vehicle innovation and public funding projects focussed on development of new processes/methods/tools for the application/product lifecycle management, development of next-generation electrified powertrain, driver assistance and telematics functionalities, definition of next-generation safe and secure EE architectures in compliancy with functional safety and cybersecurtiy norms. Since 2016 he represents FCA in Cybersecurity Task Force of ACEA WG-CONNECT.