Towards the Formal Verification of SysML v2 Models

Vince Molnár
Bence Graics
András Vörös
Department of Artificial Intelligence and Systems Engineering, Budapest University of Technology and Economics
Budapest, Hungary
{molnarv,graics,vori}@mit.bme.hu

Stefano Tonetta
Luca Cristoforetti
Fondazione Bruno Kessler
Trento, Italy
{tonettas,cristofo}@fbk.eu

Greg Kimberly
The Boeing Company
Seattle, Washington, USA
greg.kimberly@boeing.com

Pamela Dyer
Kristin Giammarco
Naval Postgraduate School
Monterey, USA
{pamela.dyer,kmgiamma}@nps.edu

Manfred Koethe
88solutions Corporation
San Diego, USA
koethe@88solutions.com

John Hester
Jamie Smith
Imandra
Austin, Texas, USA
{john,jamie}@imandra.ai

Christoph Grimm
University of Kaiserslautern
Kaiserslautern, Germany
cgrimm@rptu.de

ABSTRACT

Systems Modeling Language (SysML) is the de facto standard in the industry for modeling complex systems. SysML v2 is the new version of the language with reworked fundamentals. In this paper, we explore how the new formal semantics of SysML v2 can enable formal verification and various forms of automated reasoning. Formal verification involves mathematically proving the correctness of a system's design with respect to certain specifications or properties. This rigorous approach ensures that models behave as intended under all possible conditions. Through a detailed examination, we demonstrate how five specific tools – Gamma, MP-Firebird, Imandra, SAVVS, and SysMD – can formally analyze SysML v2 models. We show how these tools support the different concepts in the language, as well as the set of features and technologies they provide to users of SysML v2, such as model checking, theorem proving, contract-based design, or automatic fault injections. We propose a workflow for applying formal methods on SysML v2 models, illustrated by example models and artifacts generated by the above tools.

CCS CONCEPTS

• **Software and its engineering** → **System modeling languages**; **Formal methods**.

KEYWORDS

SysML v2, systems modeling, formal methods, verification and validation, automated reasoning, tools

ACM Reference Format:
Vince Molnár, Bence Graics, András Vörös, Stefano Tonetta, Luca Cristoforetti, Greg Kimberly, Pamela Dyer, Kristin Giammarco, Manfred Koethe, John Hester, Jamie Smith, and Christoph Grimm. 2024. Towards the Formal Verification of SysML v2 Models. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3652620.3687820

1 INTRODUCTION

Model-based Systems Engineering (MBSE) has been promising the automation of systems engineering workflows from the beginning, which would be especially beneficial in Verification and Validation (V&V) [21, 24, 25, 33]. However, popular MBSE languages like SysML generally lacked the capabilities required to achieve this vision. SysML's main limitations include the lack of precise semantics [16]; poor interoperability between tools, as well as between SysML and other engineering languages [9, 26]; and unsatisfactory cohesion between different diagrams representing the same information [3].

SysML v2 promises to address most of these limitations [29]. It has a formal semantics based on first-order logic, with a language Core (technically part of the Kernel Modeling Language, or KerML, which is the base language of SysML v2) that provides a formal base ontology. It enables the combination of these basic concepts and relations into more complex, high-level elements like parts, steps, or states of a system. The new language has a standardized textual syntax in addition to the graphical diagrams, which greatly simplifies interoperability and allows easier integration of external tools to process the models written in the language. Furthermore, SysML v2 features a richer set of elements, including new concepts like analysis or verification cases, formal requirements, and different kinds of specialization relationships uniformly applicable to almost all elements.

These powerful new features greatly facilitate the application of *formal methods* (FM) – in particular, *formal verification*, *constraint propagation*, *automated reasoning*, *test and code generation*, *simulation*, as well as *fault injection and safety analysis*. The Formal Methods Working Group of the OMG's Systems Modeling Community[1] was founded by organizations actively working on integrating formal methods tools with SysML v2. The group's goals include 1) providing feedback about the syntax and semantics to the language designers, 2) creating domain-specific libraries to introduce formal concepts into the language (like temporal logic), and 3) proposing a uniform and generic workflow for the application of formal methods tools. In this paper, we aim to present the current status of our effort, with a special emphasis on the proposed workflow 3) and the tools currently being integrated with the language. Even though this is still a work in progress, we believe that seamless integration of formal methods in MBSE to automate and enhance V&V workflows is essential for the success of these techniques (both MBSE and FM). Thus, we find it very important to raise awareness, get feedback, and spark more participation in the scientific community.

The paper is structured as follows. Section 2 summarizes the most important new features of SysML v2 relevant to FM. Section 3 explains the basic concepts of formal methods regarding typical use cases in systems engineering. Section 4 presents the tools currently being integrated with SysML v2 by members of the Formal Methods Working Group. Section 5 introduces the proposed workflow for applying formal methods on SysML v2 models. Finally, Section 6 concludes the paper and sketches the envisioned future work.

2 CURRENT STATE OF SYSML V2

This section summarizes the relevant new features of SysML v2 [29].

2.1 Language Architecture and Semantics

In contrast with the previous version, SysML v2 is not built on top of UML as a profile. Instead, a new general-purpose modeling language has been designed to serve as the foundation of domain-specific languages. This language is called the Kernel Modeling Language (KerML) [27].

KerML has three layers. The *Root layer* provides a graph structure for the syntax of the language with *elements* and *relationships* (which are themselves elements), as well as *annotations* that can be used to provide metainformation for elements, including language annotations and tool-specific information that can help in the integration of external tools and languages. The *Core layer* introduces the primitives of the ontological semantics: *types* divided into *classifiers* and *features*, and several relationships between them, including several *specialization* relationships. This layer defines the semantics of modeling elements in first-order logic axioms. Finally, the *Kernel layer* defines a rich set of reusable concepts (including a native expression and action language) specializing the base types and relationships and defining their semantics with constraints. These concepts are further refined in SysML v2 to provide the set of concepts typically used in systems engineering.

The ontological semantics means that KerML and SysML models are interpreted in terms of classification, i.e., things in the real or imagined *universe* the model is about are classified by some of the types in the model, in accordance with the rules defined by relationships and constraints between the types. For example, if type A specializes type B, then any valid interpretation that classifies something as an instance of type A must also classify it as an instance of type B.

As mentioned above, types are divided into classifiers and features. The former classify "single things" in the universe, while the latter classifies "sequences" (or tuples) of things (i.e., directed relationships). Since features are also types, it is possible to have longer sequences representing, e.g., navigation chains, to support modeling based on the context instead of only based on types (called "usage-oriented" modeling). Specialization relationships may be between classifiers (*subclassification*), features (*subsetting* to specify inclusion of feature values and *redefinition* to refine the constraints on an inherited feature), or between a feature and a type (*feature typing* to denote the type of feature values).

This kind of semantics is called *declarative* as it declares the constraints to accept an interpretation as valid. In contrast, an operational semantics prescribe the steps to come up with a valid interpretation. Consequently, a declarative semantics makes it easy to check if an interpretation is correct with respect to a model, but does not aid in deriving such an interpretation, whereas an operational semantics has the opposite advantages and disadvantages.

SysML v2 comes with both a textual and a graphical concrete syntax. The textual syntax has a number of clear benefits when it comes to interoperability with external tools, as it can be used a standardized and tool-independent exchange format. It is also easier to represent results produced by an external tool in SysML v2. The new Systems Modeling API and Services component provides full model access and supports shared repositories with Git-like semantics [28]. It supports shared modeling sessions, integration of external tools like CAD and PDM systems, and provides another avenue to integrate formal methods tools.

2.2 New Modeling Elements

SysML v2 introduced a number of new concepts that make the application of formal methods easier.

Requirements. Even though requirements existed in SysML v1, they only had a natural language representation and some links to other modeling elements. In addition to the natural language representation, requirements in SysML v2 may declare their subject, have attributes, and define a constraint to formally state the requirement. Furthermore, requirements may specialize each other, leading to increased reusability and a native support for requirement templates.

Analysis and verification cases. In addition to use cases, SysML v2 introduced the generic "case" concept as a usage of a subject in a

[1]Established by the Object Management Group to address open and future issues related to the SysML v2 language – see https://www.omg.org/communities/systems-modeling-community.htm.

specific context (in the case of use cases, an illustration of some user interaction). Two new cases are *analysis* and *verification cases*, which can also declare their subjects, as well as an objective (such as the calculation of a measure or the verification of a requirement) and a series of steps to derive one of its quantitative (analysis case) or qualitative (verification case) properties. These cases can also specialize each other, e.g., to refine the steps and model a more specific execution of the case.

Annotations. There are several kinds of annotations that can label model elements without contributing to the semantics. Notable examples are "doc" comments, which provide documentation, *language annotations*, which give the representation of an element in another language, and *metadata*, which can capture any kind of metainformation such as tool-specific data.

Language extension and libraries. It is very easy to create model libraries in SysML v2 due to the flexible language and rich set of specialization relationships. Furthermore, the language has a powerful mechanism for language extension that replaces stereotypes and profiles in favor of *semantic metadata* to introduce user-defined keywords, the semantics of which can be defined via an implicit specialization of the selected library concept. This solution allows for extending the language relatively seamlessly.

2.3 Standardization status

At the time of writing this paper, SysML v2 is in its last finalization phase, expected to be published in about 6-9 months. The Systems Modeling Community (SMC) is actively supporting this effort by working on parts of the language and analyzing and exercising it in different contexts and use cases to find potential issues and enhancements.

One particular use case is the language's ability to support the application of formal methods, which is investigated by the Formal Methods Working Group, including the authors of this paper. The results presented in the later sections, therefore, represent a work in progress aiming to shape the language and propose practical patterns, potentially included in the final version. Some of the authors are actively involved in the finalization of the standards, which makes this goal realistic.

Although the semantics of SysMLv2 is rooted in formal methods with a mapping to first order logic, we mentioned that these foundations have been created with an ontology-based representation of the models. The introduction of time and the semantics of executions of the models is still under development. The semantics of state machines and other behavioral models does not match easily the transition systems usually used by formal methods such as model checking [37]. One of our main goals is therefore to adjust the semantics to ease the application of formal methods.

3 FORMAL METHODS FOR SYSTEMS ENGINEERING

The language characteristics outlined in the previous section facilitate the integration of formal methods to automate the analysis, verification, and validation of models. The literature on formal methods exemplifies a variety of such integrations to provide system and software engineers with powerful methods such as formal verification, constraint propagation, automated reasoning, test and code generation, simulation, as well as fault injection and safety and reliability analysis.

The Formal Methods Working Group of the OMG's Systems Modeling Community was founded by organizations actively working on integrating formal methods tools with SysML v2. Potential use cases of such methods by software and system engineers include:

- Formalization and validation of requirements: requirements can be formalized with logical formulas; these can be expressed in SysML v2 constraints or extensions defined with libraries to introduce temporal logic operators; their consistency can be checked with SAT or SMT solvers, or with model checkers in the case of temporal logics such as LTL.
- Formal verification of behavioral models: the behaviors of the system and components can be designed with state machines and activities; formal methods such as model checking and theorem proving are able to provide proof of their correctness with respect to the requirements.
- Contract-based compositional reasoning: systems are usually decomposed into subsystems and components; their properties can be structured into assumptions and guarantees and refined along the architectural decomposition; model checking can be used to verify their refinement.
- Design space exploration: the model parameters can be used to represent design options in the same model; SAT/SMT solvers can be used for parameter instantiation and constraint propagation, exploring different design alternatives.
- Fault injection and safety analysis: the safety and reliability of the system can be analyzed taking into account faults and their propagation; the behavioral models can be extended with faults and their effect can be analyzed with model checking techniques.
- FDIR and diagnosability: the correctness of Fault Detection, Isolation, and Recovery (FDIR) components can be analyzed to ensure the reliability of the system's tolerance to faults; to this purpose, the model can be enriched with the specification of the observables and the diagnosability of faults can be formally proved.
- Model-based test case generation: the model can be used to generate systematically test cases for the final implementation; model checking or symbolic execution techniques can be used to explore different paths of the behavioral models.

4 ANALYSIS AND VERIFICATION TOOLS FOR SYSML V2

This section presents five different tools that support various analysis and verification functionalities with formal methods for SysML v2, namely SysMD (see Section 4.1), Monterey Phoenix (Section 4.2), Gamma (see Section 4.3), Imandra (Section 4.4) and SAAVS (Section 4.5). The features of these tools are summarized in Table 1.

These tools cover the majority of the use cases presented in Section 3. In an imaginary scenario, users could use SysMD to analyze the static constraints most likely applicable to the physical design of the system. They could then go on to explore properties of the envisioned dynamic behavior with the help of Monterey Phoenix. Once they are confident about the design, they could translate the resulting model into SysML and use Gamma to map it into the input languages of different formal verification tools. Either through Gamma or using direct transpilation, Imandra could be used to prove the properties of the modeled behavior and further explore corner cases. Finally, tools integrated in SAVVS (again, either through Gamma or directly) could be used to perform compositional verification safety analysis for larger configurations of the system's components. In the following sections, we present the capabilities of the tools in more detail, then Section 5 will introduce how we envision the integration of all these techniques with SysML v2.

4.1 SysMD

SysMD [34, 35] is a SysML v2 based tool that targets the use cases documentation, elicitation of requirements, modeling of architectures, and the interactive analysis and calculation of key performances. Unlike other tools in Section 4 it does not target the verification of behavior modeled by state machines. SysMD supports the modeling and elicitation of requirements by notebook-like interface in which documentation and models can be integrated in a single Markdown document. This in particular targets to permit easier exchange of models in a value chain.

Analysis and consistency checking are supported by the support for ranges on Integers and Reals and constraint propagation. For constraint propagation, SysMD applies methods known from SAT/SMT solving. By constraint propagation, it computes an over-approximation of values that satisfy all assertions and invariants. A simple example is shown in the first package of Fig. 1.

Note that also inequalities and Boolean expressions are supported. For validation/verification, specified Requirements are exported to domain-specific languages, e.g., SystemC, and after verification or characterization, the results are imported. Thereby, SysMD leaves the specific verification for specific verification tools, but provides a basic framework for contract-based approaches in different domains.

4.2 MP-Firebird

Monterey Phoenix (MP), developed at the US Navy Naval Postgraduate School (NPS), is a language, approach, and tool for formally modeling scope-complete sets of behaviors[2]. MP was initially developed to support executable software architecture [1, 2], but was quickly applied for other types of systems involving safety, security, and time-critical behaviors including hardware, organizations and roles, operational and business processes, and other workflows. The publicly accessible MP-Firebird tool[3] provides users with a guarantee of 100% coverage of paths through the provided behavior logic up to a user-defined scope, where scope is defined as the number of event iterations. This helps technical and non-technical users formally state and refine their own cognitive perceptions of how systems of interest behave and interact, and bring subtle undocumented requirements into their field of awareness and attention.

The MP-Firebird user first captures the current understanding of a system's behavior as a set of imperative and declarative rules representing actor behaviors, interactions, and other constraints in an MP schema. Next, the user runs that schema to produce a set of event traces representing all behavior logic permitted by the rules. In fact, MP produces a complete set of all behavior combinations for the modeled system and environment that is exhaustive up to a user-defined number of event iterations, or scope, so that Jackson's small scope hypothesis that most design errors can be exposed in small examples [23] can be applied. In addition to being a workforce-accessible tool for generating scope-complete and consistent sets of example event traces, a number of theorems that have a finite number of cases to check can also be proved using MP.

Once the automatic event trace generation has occurred, the user then inspects this set of traces for behavior examples that violate design intention. MP-Firebird distinctly focuses on helping people apply formal methods to their own inner thought process, as they expose their own (cognitively internal) assumptions and errors, including errors of omission concerning missed requirements and unidentified risks. An MP-Firebird user does not need to know the assertions in advance of using the tool; rather, MP helps them discover what the assertions ought to be in the first place.

This assertion discovery process can often be conducted very productively at the operational or solution-neutral architecture level, without the design details or implementation code. "Unknowns" gradually become "knowns" as they enter the MP user's field of awareness through an iterative process of inspection of the scope-complete set of execution traces, which commonly includes expected, unexpected, acceptable, and unacceptable behaviors [18]. MP-Firebird can be used at any lifecycle stage as a verification, validation, and refutation tool by novices through experts with a need to probe for and expose undocumented expectations concerning subtle but consequential behaviors that can arise from design decisions – behaviors that meet requirements, but violate expectations.

MP-Firebird is currently being used to test SysML v2 semantics in a series of model translation experiments to aid in the identification of subtle language requirements that are otherwise difficult to enumerate before example violations of intent are observed. Future work includes an interface between the MP language and KerML, and integration of the MP trace generator as a plug-in to SysML v2 implementation tools, to provision SysML users with the scope-complete execution trace generation capability of MP.

4.3 Gamma

The Gamma Statechart Composition Framework is an open-source[4] Eclipse-based tool suite for the component-based design and verification of reactive systems. At its core, it builds on a configurable *statechart language* (GSL) [19] and a *composition language* (GCL) [8] with precise semantics and validation rules to describe the functional behavior of standalone components, and their integration using various execution and communication modes to support the modeling of synchronous and asynchronous systems. The emergent models are automatically mapped into a low-level formalism, called *EXtended Symbolic Transition Systems* (XSTS) [20], serving as a *common formal representation* to capture reactive behavior.

Formal verification based on temporal properties is supported by mapping XSTS models and properties into the inputs of different model checker back-ends, namely UPPAAL [4], Theta [36], Spin [22] and nuXmv [11]. Note that different model checkers utilize different

[2]https://nps.edu/web/monterey-phoenix
[3]https://firebird.nps.edu/

[4]https://github.com/ftsrg/gamma

Table 1: Features of formal verification tools supporting SysML v2. ✓= full support; ✓̸= experimental

	Component integration	State machines	Process models	Simulation	Model checking	Theorem proving	Contract-based design	Fault injection	Back-annotation	Extensible	Semantic diff	State space analysis	Diagnosability analysis	Free to use
SysMD	✓			✓̸	✓̸		✓	✓	✓	✓				✓
MP-Firebird	✓	✓	✓		✓									✓
Gamma	✓	✓	✓̸	✓̸	✓		✓̸	✓	✓	✓		✓̸		✓
Imandra	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓̸
SAVVS	✓	✓			✓		✓	✓		✓			✓	

Table 2: SysML v2 language elements supported by the tools. ✓= full support; ✓̸= experimental

	Ports	Hierarchical parts	Hierarchical states	Entry/exit actions of states	Guards on transitions	Effects on transitions	Logical expressions	Arithmetic expressions	Variable assignments	Attributes in parts	Integer types	Real types	Enumeration types	Redefinitions	Requirements	Verification Cases
SysMD	✓	✓					✓	✓	✓	✓	✓	✓	✓̸	✓	✓	
Gamma	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Imandra	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SAVVS	✓̸	✓			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		

algorithms and can fit the needs of verifying systems with various characteristics, like timed, synchronous, asynchronous, data- or control-intensive systems. The mappings feature *model reduction* and *model slicing* algorithms to allow for the verification of large-scale systems. Back-annotation facilities automatically map the verification results to execution traces in a high-level *trace language* (GTL) [8]. Building on the model checking and back-annotation functionalities [17], Gamma generates *integration tests* (in the form of execution traces) based on manually defined test targets captured by declarative model queries [7], as well as customizable structural (model element based), dataflow-based and interactional coverage criteria [20]. Test cases are *optimized* and concretized to different platforms, e.g., C or Java, to detect faults in the implementation of components, such as missing implementation of states or transitions, incorrect variable definitions and uses or wrong implementation of component interactions. Gamma also supports *fault injection* based *safety assessment* via the xSAP tool [10] in addition to *contract-based design* based on model checking emergent model behaviors against LSC scenario descriptions [5, 6], and LTL properties using the OCRA tool [12].

Gamma features an automated model transformation from the SysML v2 Pilot Implementation that maps (hierarchical) part definitions with state-based behavior into GCL and GSL models. The emergent Gamma models then can undergo formal verification using the above-described facilities. The results are automatically back-annotated to SysML v2 using two different representations (namely event sequence representations and process models) highlighting different aspects of the observed execution traces.

4.4 Imandra

Imandra is a cloud-native automated reasoning platform [30] designed for the analysis of algorithms. Imandra is used for formal verification, state space analysis, test-case generation, constraint solving, rule synthesis, and other symbolic AI applications across finance, defense, automotive, and manufacturing industries [14, 15, 31, 32]. Imandra Modeling Language (IML) is a general purpose programming language (a formalized subset of the open source programming language OCaml) designed for specifying models and programs that can be reasoned about with Imandra.

Imandra’s logic is computational, based on a pure subset of OCaml, with restrictions on types and higher-order functions that allow conjectures to be translated into multi-sorted first-order logic with theories, including arithmetic and datatypes [30]. Imandra has novel features supporting large-scale industrial applications, including a seamless integration of bounded and unbounded verification (e.g., model checking and induction), first-class computable counterexamples and efficiently executable models. The core reasoning mechanisms of Imandra are 1) a semi-complete procedure for finding models of formulas in the logic mentioned above, centered around the lazy expansion of recursive functions, and 2) an inductive waterfall and simplifier which "lifts" many Boyer-Moore ideas to Imandra's typed higher-order setting. These mechanisms are tightly integrated and subject to many forms of user control.

Imandra has recently added a SysML-v2-to-IML transpiler to enable the formal analysis and verification of SysML v2 models. The transpiler converts SysML v2 models to native IML, and Imandra's automated reasoning can then be brought to bear on the IML encoding of the model semantics. This leverages the dual nature of IML, expressing SysML models naturally as executable OCaml programs, which in turn have formal semantics by virtue of Imandra's mechanized formal semantics of IML. In addition to facilitating precise reasoning about the model, this dual nature allows the IML models to be used as high-performance (formal) digital twins, facilitating runtime monitoring, audits, and optimization.

Imandra's SysML integration is also powering new generative AI applications, including a hallucination-free natural language interface combining LLMs with Imandra's automated reasoning for answering questions about SysML models and their possible behaviors. This empowers Model-based Systems Engineers to leverage formal methods without learning a new language or having a deep understanding of automated reasoning, with all results derived by Imandra reasoning about the SysML model via the SysML-v2-to-IML transpiler.

4.5 SAVVS

SAVVS (Safety Analysis, Validation and Verification for SysMLv2) is a set of plugins to extend the Eclipse-based SysMLv2 Pilot Implementation with the purpose of enabling validation, formal verification, and safety analysis of formal models. SAVVS can be seen as the evolution of CHESS [13], a cross-domain, model-driven, component-based and open-source tool for the development of high-integrity systems, which supports SysML v1.

SAVVS features an automatic Model-to-Model transformation, eased by the use of open-source FBK libraries and facilitates the use of FM tools OCRA [12], nuXmv [11], and xSAP [10]. The SysMLv2 language has been enriched with ad-hoc libraries to support LTL operators, fault injection, and terms to exploit the contract-based methodology.

The integration with nuXmv enables the verification of SysMLv2 state machines with powerful state of the art SMT-based model checking algorithms that combine search-based techniques with automated deduction and abstraction refinement.

Thanks to the integration with OCRA, SAVVS provides validation of assumption/guarantee properties in contracts as well as checks on contract refinements and composite contract implementations exploiting the model checker. Model checking of state machines can be therefore scaled up by analyzing the system model compositionally, exploiting the contract-based decomposition.

Finally, fault injection enables Fault Tree Analysis and Failure Mode and Effect Analysis. SAVVS uses xSAP as backend to generate fault trees and FMEA tables. A partial support to Fault Detection and Isolation provides Diagnosability Analysis and generation of minimum observables set.

Counterexample traces, fault trees, and FMEA tables are reported in the tool by means of dedicated views, as can be seen in Figure 8 and Figure 9.

5 UNIFIED VERIFICATION WORKFLOW

This section proposes a unified way of handling various V&V tasks in SysML v2, outlining how the tools presented in Section 4 integrate (or plan to integrate) their results into the *unified verification workflow*, developed by experts of the respective tools.

We will use four examples. The *Tank* model can be seen in the first package of Figure 1, and it models a simple tank with a height, width, and length, which can be used to calculate its volume. There is also a requirement restricting the volume. The tank's dimensions are also bounded by ranges, which are currently displayed in SysMD's notation (this feature is not yet available in SysML v2).

The *LightSwitch* model (depicted in the rest of Figure 1) is a simple state-based component that has a port, through which "signals" can be sent to turn the switch on or off (SysML v2 does not have specific modeling elements for signals, but *items* can be used to model things that can be exchanged). The state machine has two states, and two Boolean variables are kept in sync with the states via the *entry* and *exit* actions of the *on* state.

The third model is not included, but it is a more complex model of a traffic light that we will use to demonstrate Imandra's semantic analysis capabilities.

The fourth model, also not included for space limits, is a redundant sensor example, which exhibits a simple redundancy schema and is used to demonstrate the safety analysis and fault tree generation with SAVVS and xSAP (Figure 8).

5.1 Analysis and Verification Tasks

In Figure 3, we propose to use analysis and verification cases from SysML v2 to model the analysis and verification tasks directly in SysML. This approach has a number of advantages:

- The description of the task can be included in the model repository, which will provide version control for the configuration of these tasks.
- Results of the analyses can also be included in the model as refinements (via specialization) of the original task (see Section 5.2).
- V&V tasks and their results can refer to and may be referred to from other parts of the model, enhancing traceability by justifying design decisions with analyses and their results.
- Describing tasks, configurations, and results in SysML provides a tool-independent way of managing this information, facilitating interoperability.
- Tasks in the model can be executed automatically in continuous integration workflows by querying the model and passing the tasks to the appropriate tools.

The tasks and results defined in Figures 3 and Figure 4 use elements from a *formal methods model library* which we plan to propose as a standard library. For now, an illustration is given in Figure 2. It contains 1) illustrations of using library concepts, such as the special objective for test generation, to model configurations of tasks; 2) an example for user-defined keywords, such as the one for test generation tasks, which are special verification cases (see [29] for a detailed explanation of the language extension mechanism); 3) a stub of a temporal logic library; and 4) a metadata definition that can be used to annotate the model with information that is represented by external artifacts.

```
package TankModel {
    import SI::*;
    import ISQ::*;
    part def Tank {
        attribute height: LengthValue = [10.0 .. 100.0] [cm];
        attribute width: LengthValue = [1.0 .. 1.1] [m];
        attribute length: LengthValue = [1.0 .. 1.1] [m];
        attribute volume: VolumeValue = height * width * length;
    }
    requirement def VolumeRange {
        subject t : Tank;
        require constraint maxVol { t.volume <= 2000.0 [L] }
        require constraint minVol { t.volume >= 1000.0 [L] }
    }
}
package LightSwitchModel {
    import ScalarValues::Boolean;
    item def TurnOn;
    item def TurnOff;
    port def Control {
        in item turnOn : TurnOn;
        in item turnOff : TurnOff;
    }
    part def LightSwitch {
        port control : Control;
        attribute isOn : Boolean := false;
        attribute isOff : Boolean := false;
        exhibit state Status {
            state off;
            state on {
                entry action { assign isOn := true;
                    assign isOff := false; }
                exit action { assign isOn := false;
                    assign isOff := true; }
            }
            entry; then off;
            transition first off
                accept turnOn : TurnOn via control
                then on;
            transition first on
                accept turnOff : TurnOff via control
                then off;
}   }   }
```

Figure 1: Tanks and Switch example models in SysML v2.

```
package FormalMethods {
    // Test generation objective
    enum def CoverageKind {
        State; Transition; Interaction; //...
    }
    requirement def TestGenerationObjective {
        attribute coverage : CoverageKind default State;
    }
    // User-defined keyword for #testgeneration tasks
    abstract verification def TestGeneration {
        objective : TestGenerationObjective;
    }
    abstract verification tgs : TestGeneration[*] nonunique;
    metadata def testgeneration :> Metaobjects::SemanticMetadata {
        :>> baseType = tgs meta SysML::Usage;
    }
    // Temporal logic (real library under development)
    calc def <MustAlways> AG;
    //...
    // External artifacts as metadata
    metadata def ExternalArtifact {
        :> annotatedElement : SysML::CaseDefinition;
        attribute name : ScalarValues::String;
        attribute uri : ScalarValues::String;
}   }
```

Figure 2: Illustration of the envisioned formal methods model library.

```
package TankAnalysis {
    import TankModel::*;
    analysis def CalculateDimensions {
        subject t : Tank;
        objective : VolumeRange;
}   }
package LightSwitchAnalysis {
    import LightSwitchModel::*;
    import FormalMethods::*;
    verification def VerifyProperty {
        subject ls : LightSwitch [1];
        objective proveCorrectFlags {
            verify requirement {
                require constraint {
                    MustAlways( ls.isOn xor ls.isOff )
    }   }   }   }
    #testgeneration def GenerateTests {
        subject ls : LightSwitch [1];
        objective : TestGenerationObjective {
            :>> coverage = CoverageKind::Transition;
    }   }
    verification def GenerateFaultTree {
        subject ls : LightSwitch [1];
        objective o : FaultTreeGenerationObjective {
            :>> faultInjection = Injections::FaultyWrites;
            :>> topLevelEvent { ls.inOn and ls.isOff }
}   }   }
```

Figure 3: Analysis and verification cases describing formal methods tasks.

The tasks in Figure 3 include 1) an analysis case to apply constraint propagation to the tank model, with respect to the requirement specified for its volume; 2) a verification task describing the verification of a temporal property[5] requiring the two Boolean flags to be in an *exclusive or* relationship; 3) a test generation task that also uses a verification case, but it is declared with the user-defined keyword `#testgeneration` and its objective must be specified with the `TestGenerationObjective` library element that can be parameterized with the kind of coverage the test generator has to maximize; and 4) a fault tree generation task that defines the automatic fault injection strategy, as well as the top level event – the question is, what combination of faults can lead to the satisfaction of the constraint describing the top-level event.

5.2 Representation of Analysis Results

As mentioned among the benefits of modeling tasks in SysML v2, many of the results can also be modeled in SysML, which shall facilitate traceability and interoperability between tools. Here, we illustrate how we envision the description of different results, based on the outputs produced by the tools presented in Section 4. The overarching pattern is that results are modeled as refinements (specializations) of the tasks, *redefining* (denoted by :>>) the subject to introduce new details inferred by the formal methods tools (such as stricter ranges, one or more execution traces, etc.).

[5]Note that at the time of writing this paper, temporal logic is not yet supported in KerML or SysML v2, but the Formal Methods Working Group is working on a proposal to include it as a library.

Constraint Propagation. The results of constraint propagation performed by SysMD can be modeled as a refinement of the original task (see `Result :> CalculateDimensions`) by redefining the subject and its attributes, narrowing the ranges. In this example, SysMD computes the height as $[82.6..100]cm$, and the possible volume as $[1000..1210]l$, considering the `VolumeRange` requirement.

Representation of Traces. An execution trace can also be modeled as a refinement of the original model. The models in `LightSwitchCases` illustrate this approach to represent the results of the verification and the test generation tasks. In `Witness` (see Figure 4), we see a counterexample for the requirement specified in the in-line unnamed requirement element inside the `proveCorrectFlags` objective in Figure 3. The subject is redefined, along with its exhibited state, and a *subset* (denoted by :>) of the `off` state is declared as `s1`. Since SysML has ontological semantics based on classification, the subset relationship indeed expresses our intent because `s1` will classify some of the things (system states in the real world) classified by `off` – in this case, the first period when the switch is off. Since there is currently no concise way to express the values of attributes while the system is in a given state, we include this additional information as a comment – which demonstrates that when we first reach the `off` state, the flags are not (yet) set correctly because that will be handled only by the entry and exit actions of the on state.

A more elaborated execution trace is generated as the single test case found by Gamma to cover all the transitions in the model, represented by `Test1`. A similar output could describe the behaviors found by MP during the exploration of the allowed behaviors, which is currently visualized as illustrated in Figure 5. This model is again a refinement of the original task, and it represents a test case, similarly detailing the expected evolution of the state machine through the redefinition of the subject and its exhibited state, as well as the inputs necessary to trigger this behavior. The item definitions `TurnOn1` and `TurnOff1` specialize the corresponding item definitions from the original model provided by the user, representing a subset of their instances – those that are sent in this execution. These more special items are used in the redefinition of the transitions, which follow the same pattern of subsetting the original transition as the states, as well as the send actions describing the test case itself.

Representation of Auxiliary Information. Some analysis tools like Imandra can calculate auxiliary information about the state space. These are generally better suited to visual representations. Currently, SysML v2 has an open-source implementation based on Jupyter [3], which is also used by Imandra to allow users to generate visualizations of their model's state space. Illustrations of these artifacts are shown in Figures 6 and 7, which show the 174 invariant behavior regions of the traffic light model in an interactive, explorable map, as well as a semantic diff of the state space before and after a change in the model, with the two regions that actually changed. These auxiliary artifacts can be included in the model with a URI, as illustrated in the `Witness` in Figure 4.

Another example of auxiliary artifacts is the fault tree and FMEA table generated by SAVVS. While these could be modeled in SysML via a domain-specific library (which is subject to future work), it is often better to visualize them. Figure 8 and Figure 9 show the output of SAVVS for the redundant sensor example model.

```
package TankResults {
    import TankAnalysis::*;
    analysis def Result :> CalculateDimensions {
        subject t { // Values computed by constraint propagation
            :>> height = 82.6 .. 100.0 [cm];
            :>> width  = 1.0 .. 1.1 [m];
            :>> length = 1.0 .. 1.1 [m];
            :>> volume = 1000 .. 1210 [L];
}   }   }
package LightSwitchTraces {
    import LightSwitchAnalysis::*;
    import VerificationCases::*;
    verification def Witness :> VerifyProperty {
        // Witness execution trace
        subject ls : LightSwitch [1] {
            :>> Status {
                state s1 :> off; // isOn = false, isOff = false;
        }   }
        @ExternalArtifact{name = "State Space Map";
                uri = "<link to external artifact>";}
        return verdict : VerdictKind = VerdictKind::fail;
    }
    #testgeneration def Test1 :> GenerateTests {
        // Expected execution trace
        item def TurnOn1 :> TurnOn;
        item def TurnOff1 :> TurnOff;
        subject ls : LightSwitch [1] {
            :>> Status {
                state s1 :> off; // isOn = false, isOff = false;
                transition first s1
                    accept turnOn : TurnOn1 via control
                    then s1;
                state s2 :> on; // isOn = true, isOff = false;
                transition first s2
                    accept turnOff : TurnOff1 via control
                    then s3;
                state s3 :> off; // isOn = false, isOff = true;
        }   }
        // Test execution (inputs)
        send TurnOn1() to sw.control;
        then send TurnOff1() to sw.control;
    }
    verification def FaultTree :> GenerateFaultTree {
        @ExternalArtifact{uri = "<link to fault tree>";}
    }
}
```

Figure 4: Constraint propagation, model checking and test generation results of the analysis tasks in SysML.

6 CONCLUSION

In this paper, we presented how the new SysML v2 language can be integrated with formal methods techniques and tools via its new features. We introduced five tools that are already being integrated with the language, and illustrated how we envision the representation of their inputs and outputs. We proposed to model the V&V-related tasks and their results in SysML v2 to achieve a tool-independent, versionable, and traceable representation that seamlessly fits into the system model.

While at the time of writing, SysML v2 is not yet released, and the Formal Methods Working Group (including most authors) is actively working on the integration of formal methods and the new language, we hope our work can inspire both the MBSE and FM communities to investigate this topic and contribute with novel ideas. In the future, we will continue the integration of our tools along the lines of the solutions proposed in this paper, gradually building a

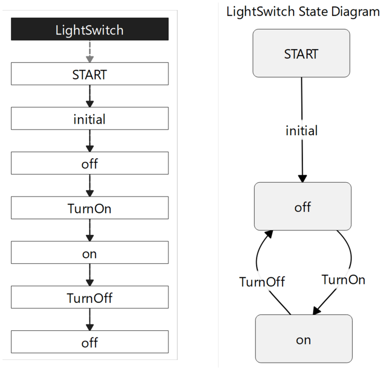
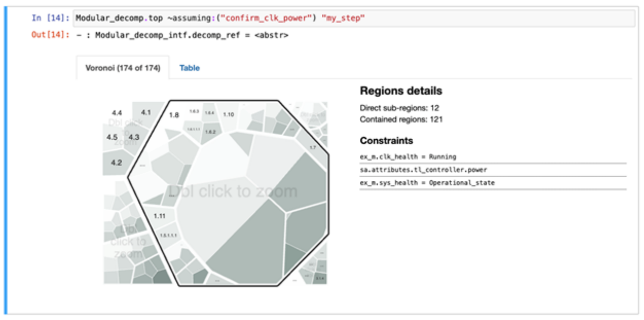


Figure 5: An example trace (left) from the set of 3 traces (when run at scope 1) that then generates the state diagram (right) for the MP LightSwitch model. A state diagram is built in MP by running a user’s model code. If the code has flaws, the traces will display those flaws, and the output state diagram will thus be incorrect until the issues are fixed.



```
In [14]: Modular_decomp.top ~assuming:("confirm_clk_power") "my_step"
Out[14]: - : Modular_decomp_intf.decomp_ref = <abstr>
```

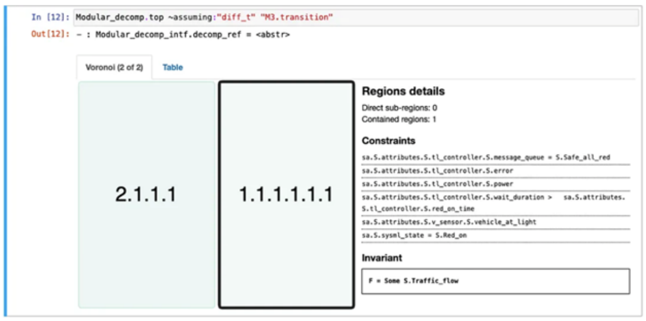
Regions details

Direct sub-regions: 12
Contained regions: 121

Constraints

```
ex_m.clk_health = Running
sa.attributes.tl_controller.power
ex_m.sys_health = Operational_state
```

Figure 6: Example output for Imandra’s invariant behavior regions functionality.



```
In [12]: Modular_decomp.top ~assuming:"diff_t" "M3.transition"
Out[12]: - : Modular_decomp_intf.decomp_ref = <abstr>
```

Regions details

Direct sub-regions: 0
Contained regions: 1

Constraints

```
sa.S.attributes.S.tl_controller.S.message_queue = S.Safe_all_red
sa.S.attributes.S.tl_controller.S.error
sa.S.attributes.S.tl_controller.S.power
sa.S.attributes.S.tl_controller.S.wait_duration >  sa.S.attributes.
S.tl_controller.S.red_on_time
sa.S.attributes.S.v_sensor.S.vehicle_at_light
sa.S.sysml_state = S.Red_on
```

Invariant

```
F = Some S.Traffic_flow
```

Figure 7: Example output for Imandra’s semantic diff functionality.

standardized formal methods model library and identifying best practices to include analysis-related information in the model.

ACKNOWLEDGMENTS

We would like to thank every member of the Formal Methods Working Group who helped us write this paper with their insights and comments. We are also grateful to Ed Seidewitz and Conrad Bock, who never get tired of answering our questions. SysMD received support (GENIAL, KI4BoardNet) from the German BMBF. Gamma received support project no. 2022-1.2.4-EUREKA-2023-00013 under the 2022-1.2.4-EUREKA funding scheme.

REFERENCES

[1] Mikhail Auguston. 2009. Monterey Phoenix, or how to make software architecture executable. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming systems languages and applications.* ACM, 1031–1040.
[2] Mikhail Auguston. 2009. Software architecture built from behavior models. In *ACM SIGSOFT Software Engineering Notes*, Vol. 34. 1–15.
[3] Manas Bajaj, Sanford Friedenthal, and Ed Seidewitz. 2022. Systems Modeling Language (SysML v2) Support for Digital Engineering. *INSIGHT* 25, 1 (2022), 19–24. https://doi.org/10.1002/inst.12367 arXiv:https://incose.onlinelibrary.wiley.com/doi/pdf/10.1002/inst.12367
[4] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Håkansson, Paul Pettersson, Wang Yi, and Martijn Hendriks. 2006. Uppaal 4.0. In *Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems (QEST ’06).* IEEE Computer Society, USA, 125–126. https://doi.org/10.1109/QEST.2006.59
[5] Bence Graics, Vince Molnár, and István Majzik. 2021. Contract-Based Specification and Test Generation for Adaptive Systems. In *16th International Conference on Dependability of Computer Systems (DepCoS-RELCOMEX)*, Vol. 1389. Springer, 136–145. https://doi.org/10.1007/978-3-030-76773-0
[6] Bence Graics, Vince Molnár, and István Majzik. 2023. Component-based specification, design and verification of adaptive systems. *Systems Engineering* 26, 5 (2023), 567–589. https://doi.org/10.1002/sys.21675
[7] Bence Graics, Vince Molnár, and István Majzik. 2023. Configurable Model-Based Test Generation for Distributed Controllers Using Declarative Model Queries and Model Checkers. In *Formal Methods for Industrial Critical Systems*, Alessandro Cimatti and Laura Titolo (Eds.). Springer Nature Switzerland, Cham, 76–95. https://doi.org/10.1007/978-3-031-43681-9_5
[8] Bence Graics, Vince Molnár, András Vörös, István Majzik, and Dániel Varró. 2020. Mixed-Semantics Composition of Statecharts for the Component-Based Design of Reactive Systems. *Software and Systems Modeling* 19 (2020), 1483–1517. https://doi.org/10.1007/s10270-020-00806-5
[9] Mary Bone, Mark Blackburn, Benjamin Kruse, John Dzielski, Thomas Hagedorn, and Ian Grosse. 2018. Toward an Interoperability and Integration Framework to Enable Digital Thread. *Systems* 6, 4 (2018). https://doi.org/10.3390/systems6040046
[10] M. Bozzano, A. Cimatti, M. Gario, D. Jones, and C. Mattarei. 2021. Model-based Safety Assessment of a Triple Modular Generator with xSAP. *Formal Aspects of Computing* 33, 2 (2021), 251–295. https://doi.org/10.1007/s00165-021-00532-9
[11] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. 2014. The nuXmv Symbolic Model Checker. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 334–342.
[12] Alessandro Cimatti, Michele Dorigatti, and Stefano Tonetta. 2013. OCRA: A Tool for Checking the Refinement of Temporal Contracts. In 28th *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Ewen Denney, Tevfik Bultan, and Andreas Zeller (Eds.). IEEE, 702–705.
[13] Alberto Debiasi., Felicien Ihirwe., Pierluigi Pierini., Silvia Mazzini., and Stefano Tonetta. 2021. Model-based Analysis Support for Dependable Complex Systems in CHESS. In *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development - MODELSWARD.* INSTICC, SciTePress, 262–269. https://doi.org/10.5220/0010269702620269
[14] Remi Desmartin, Grant O. Passmore, and Ekaterina Komendantskaya. 2022. Neural Networks in Imandra: Matrix Representation as a Verification Choice. In *Proc. 5th Int. Workshop of Software Verification and Formal Methods for ML-Enabled Autonomous Systems (FoMLAS) and 15th Int. Workshop on Numerical Software Verification (NSV).* 78–95.
[15] Remi Desmartin, Grant O. Passmore, Ekaterina Komendantskaya, and Matthew Daggit. 2022. CheckINN: Wide Range Neural Network Verification in Imandra. In *Proc. 24th Int. Symposium on Principles and Practice of Declarative Programming (PPDP).* 3:1–3:14.
[16] Márton Elekes, Vince Molnár, and Zoltán Micskei. 2023. Assessing the specification of modelling language semantics: a study on UML PSSM. *Software Quality Journal* 31, 2 (mar 2023), 575–617. https://doi.org/10.1007/s11219-023-09617-5
[17] Gordon Fraser, Franz Wotawa, and Paul E. Ammann. 2009. Testing with model checkers: a survey. *Software Testing, Verification and Reliability* 19, 3 (2009), 215–261. https://doi.org/10.1002/stvr.402
[18] Kristin Giammarco. 2024. *System Behavior Specification Verification and Validation (V&V).* John Wiley & Sons, Hoboken, NJ, 219–240.

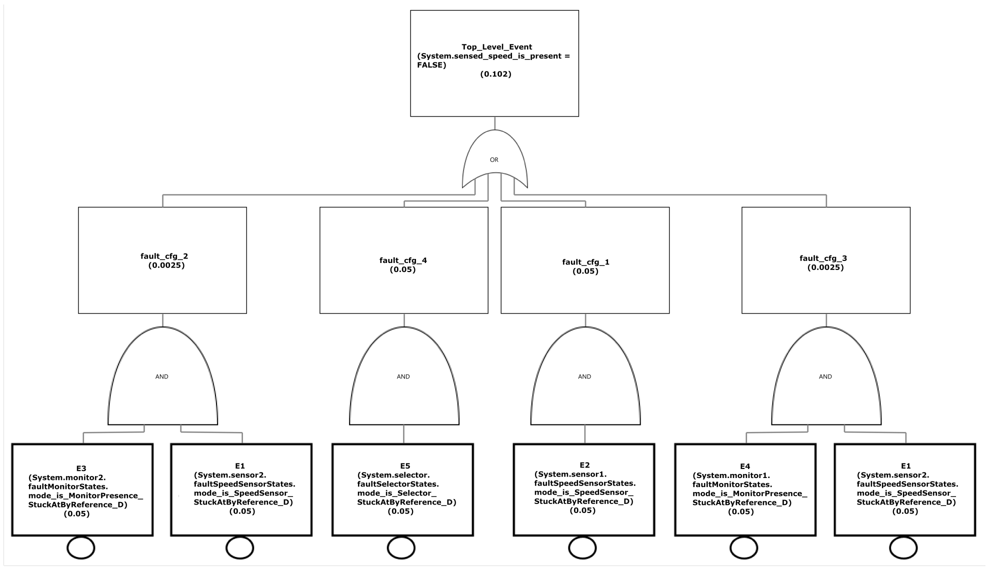
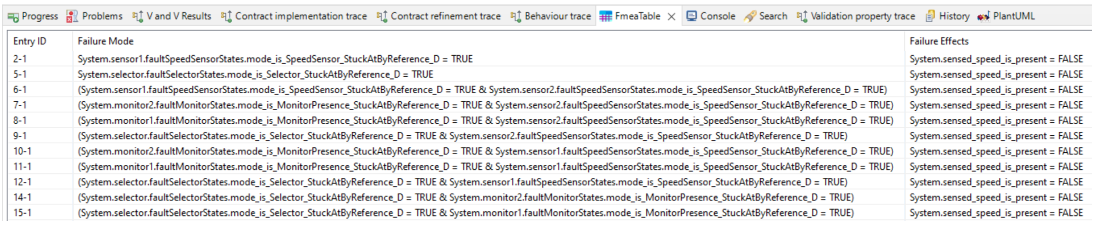


Figure 8: Example fault tree generated by SAVVS.



Entry ID	Failure Mode	Failure Effects
2-1	System.sensor1.faultSpeedSensorStates.mode_is_SpeedSensor_StuckAtByReference_D = TRUE	System.sensed_speed_is_present = FALSE
5-1	System.selector.faultSelectorStates.mode_is_Selector_StuckAtByReference_D = TRUE	System.sensed_speed_is_present = FALSE
6-1	(System.sensor1.faultSpeedSensorStates.mode_is_SpeedSensor_StuckAtByReference_D = TRUE & System.sensor2.faultSpeedSensorStates.mode_is_SpeedSensor_StuckAtByReference_D = TRUE)	System.sensed_speed_is_present = FALSE
7-1	(System.monitor2.faultMonitorStates.mode_is_MonitorPresence_StuckAtByReference_D = TRUE & System.sensor2.faultSpeedSensorStates.mode_is_SpeedSensor_StuckAtByReference_D = TRUE)	System.sensed_speed_is_present = FALSE
8-1	(System.monitor1.faultMonitorStates.mode_is_MonitorPresence_StuckAtByReference_D = TRUE & System.sensor2.faultSpeedSensorStates.mode_is_SpeedSensor_StuckAtByReference_D = TRUE)	System.sensed_speed_is_present = FALSE
9-1	(System.selector.faultSelectorStates.mode_is_Selector_StuckAtByReference_D = TRUE & System.sensor2.faultSpeedSensorStates.mode_is_SpeedSensor_StuckAtByReference_D = TRUE)	System.sensed_speed_is_present = FALSE
10-1	(System.monitor2.faultMonitorStates.mode_is_MonitorPresence_StuckAtByReference_D = TRUE & System.sensor1.faultSpeedSensorStates.mode_is_SpeedSensor_StuckAtByReference_D = TRUE)	System.sensed_speed_is_present = FALSE
11-1	(System.monitor1.faultMonitorStates.mode_is_MonitorPresence_StuckAtByReference_D = TRUE & System.sensor1.faultSpeedSensorStates.mode_is_SpeedSensor_StuckAtByReference_D = TRUE)	System.sensed_speed_is_present = FALSE
12-1	(System.selector.faultSelectorStates.mode_is_Selector_StuckAtByReference_D = TRUE & System.sensor1.faultSpeedSensorStates.mode_is_SpeedSensor_StuckAtByReference_D = TRUE)	System.sensed_speed_is_present = FALSE
14-1	(System.selector.faultSelectorStates.mode_is_Selector_StuckAtByReference_D = TRUE & System.monitor2.faultMonitorStates.mode_is_MonitorPresence_StuckAtByReference_D = TRUE)	System.sensed_speed_is_present = FALSE
15-1	(System.selector.faultSelectorStates.mode_is_Selector_StuckAtByReference_D = TRUE & System.monitor1.faultMonitorStates.mode_is_MonitorPresence_StuckAtByReference_D = TRUE)	System.sensed_speed_is_present = FALSE

Figure 9: Example FMEA table generated by SAVVS.

[19] Bence Graics. 2016. *Documentation of the Gamma Statechart Composition Framework v0.9.* Technical Report. Budapest Univ. of Technology and Economics, Dept. of Measurement and Information Systems. https://tinyurl.com/yeywrkd6

[20] Bence Graics, Milán Mondok, Vince Molnár, and István Majzik. 2024. Model-Based Testing of Asynchronously Communicating Distributed Controllers. In *Formal Aspects of Component Software*, Javier Cámara and Sung-Shik Jongmans (Eds.). Springer Nature Switzerland, Cham, 23–44. https://doi.org/10.1007/978-3-031-52183-6_2

[21] Alex Hazle and James Towers. 2020. Good Practice in MBSE Model Verification and Validation.

[22] G.J. Holzmann. 1997. The model checker SPIN. *IEEE Transactions on Software Engineering* 23, 5 (1997), 279–295. https://doi.org/10.1109/32.588521

[23] Daniel Jackson. 2012. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, Cambridge, Massachusetts.

[24] Diego Latella, Istvan Majzik, and Mieke Massink. 1999. Towards a Formal Operational Semantics of UML Statechart Diagrams. In *Formal Methods for Open Object-Based Distributed Systems*, Paolo Ciancarini, Alessandro Fantechi, and Robert Gorrieri (Eds.). Springer US, Boston, MA, 331–347.

[25] B. Nastov, V. Chapurlat, F. Pfister, and C. Dony. 2017. MBSE and V&V: a tool-equipped method for combining various V&V strategies. *IFAC-PapersOnLine* 50, 1 (2017), 10538–10543. https://doi.org/10.1016/j.ifacol.2017.08.1309 20th IFAC World Congress.

[26] Mara Nikolaidou, George-Dimitrios Kapos, Anargyros Tsadimas, Vassilios Dalakas, and Dimosthenis Anagnostopoulos. 2015. Simulating SysML models: Overview and challenges. In *2015 10th System of Systems Engineering Conference (SoSE).* 328–333. https://doi.org/10.1109/SYSOSE.2015.7151961

[27] OMG. 2024. Kernel Modeling Language. http://www.omg.org/spec/KerML/1.0

[28] OMG. 2024. OMG Systems Modeling API and Services. http://www.omg.org/spec/SystemsModelingAPI/1.0

[29] OMG. 2024. OMG Systems Modeling Language. http://www.omg.org/spec/SysML/2.0

[30] Grant Passmore, Simon Cruanes, Denis Ignatovich, Dave Aitken, Matt Bray, Elijah Kagan, Kostya Kanishev, Ewen Maclean, and Nicola Mometto. 2020. The Imandra Automated Reasoning System (System Description). In *Automated Reasoning (Lecture Notes in Computer Science)*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.). Springer International Publishing, Cham, 464–471. https://doi.org/10.1007/978-3-030-51054-1_30

[31] Grant Olney Passmore. 2021. Some Lessons Learned in the Industrialization of Formal Methods for Financial Algorithms. In *Proc. 24th Int. Symposium on Formal Methods (FM).* 717–721.

[32] Grant Olney Passmore and Denis Ignatovich. 2017. Formal Verification of Financial Algorithms. In *Automated Deduction – CADE 26*, Leonardo de Moura (Ed.). Springer International Publishing, Cham, 26–41.

[33] Jean-François Pétin, Dominique Evrot, Gérard Morel, and Pascal Lamy. 2010. Combining SysML and formal methods for safety requirements verification. In *22nd International Conference on Software & Systems Engineering and their Applications.* Paris, France, CDROM.

[34] Sebastian Post and Christoph Grimm. 2023. Co-Design of Automotive Boardnet Topology and Architecture. In *DVCon Europe 2023; Design and Verification Conference and Exhibition Europe.* VDE-Verlag, 42–49.

[35] Axel Ratzke, Sebastian Post, Johannes Koch, and Christoph Grimm. 2024. Constructive Model Analysis of SysMLv2 Models by Constraint Propagation. In *2024 19th Annual System of Systems Engineering Conference (SoSE).* 239–244. https://doi.org/10.1109/SOSE62659.2024.10620947

[36] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. 2017. Theta: a Framework for Abstraction Refinement-Based Model Checking. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, Daryl Stewart and Georg Weissenbacher (Eds.). 176–179. https://doi.org/10.23919/FMCAD.2017.8102257

[37] Ármin Zavada and Vince Molnár. 2024. From Hard-Coded to Modeled: Towards Making Semantic-Preserving Model Transformations More Flexible. In *Proceedings of the 31th Minisymposium.* 41–45. https://doi.org/10.3311/MINISY2024-008