Integrating Formal Methods with Model-Driven Engineering

Opeyemi O. Adesina
School of Electrical Engineering and Computer Science
University of Ottawa, Ottawa, Canada
oadesina@uottawa.ca

***Abstract*—This paper presents the initial exploration work and proposal for our research to integrate formal methods with model-driven engineering. An extensive literature exists with the goal of facilitating the adoption of formal methods for educational and industrial practices, yet its adoption for teaching introductory software engineering courses and analyzing critical software systems in the industry is poor. The goal of this research is to provide an easy-to-use approach for using formal methods for industrial and academic purposes. Our approach is based on generating formal representations of static and dynamic abstractions of software expressed in a textual language, called Umple, which is derived from UML. To enrich the modeling experience, we adopt a pattern-based approach to specify various object-oriented and transition patterns. To maintain scalability of the dynamic aspects, we adopted a compositional approach to integrate hierarchical systems. To ensure correctness of our approach, we have adopted simulation and rigorous test-driven development methodologies. Current results have demonstrated that the constraints and generated formal methods code represent the patterns faithfully.**

***Index Terms*—Formal Methods, Model-Driven Engineering, Software Engineering, Model Checking, Scalability, Automated Code Generation.**

I. Introduction

As the complexity of real-world software systems grows relentlessly higher, the risk of project and system failure remains unabated. This phenomenon is domain independent, as automotive [1], health [2], avionics [3] and business [4] examples attest. Unfortunately, expecting human beings developing such systems to prevent failures by detecting faults is unreasonable unless the humans are supported by sophisticated tools. Such tools must match increasing complexity by increasing the use of abstractions with rigorous mathematical underpinnings.

Tools enabling sound mathematical analysis of software, collectively called *formal methods*, have been available for decades. However, their uptake has been slow since they tend to be too hard for all but the most accomplished computer scientists to use, tend not to scale well, and tend to be somewhat special-purpose. Another set of tools and techniques in the field called *Model-Driven Engineering* (MDE), combats complexity by allowing relatively easy specification and generation of systems, bypassing the need for humans to understand what is being generated.

The easiest-to-use modeling techniques tend not to be well integrated with state of the art formal methods. This is the issue we address in this paper. In particular, our objective is to allow developers to employ the easy-to-use modeling language technology Umple to generate systems, while delegating to state-of-the-art formal methods to transparently analyse such systems. In doing so, we hope to increase the applicability of formal methods, and hence improve the quality of software. We hope to make this technology so easy to use that formal methods can even be used 'behind the scenes' in introductory software development courses.

The rest of this paper is organized as follow. In Section II, we present the problem we observed to limit adoption of existing tools and motivation for this work. Section III is a review of relevant related work with emphasis on their similarities to and differences from our work. Section IV presents our proposed solution to the central problem addressed in this work. Section V is a summary of our initial exploratory work on integrating formal methods with MDE and results. We explicate the target contributions of this work in Section VI. In Section VII, we state our plans to verify and validate our work. Finally, we present the current status of this research.

II. Problem Statement and Motivation

In spite of the attention and potential of formal methods to guarantee bug-free software systems, its adoption for industrial and teaching purposes is poor. Formal methods are too difficult to use by ordinary developers, due to complexity, scalability and tool diversity issues. As a result, formal methods have low adoption levels.

Formal methods are used to some extent in the industries, but not universally due to the above issues. Thus, by bridging the gap, we can reduce the time to market and improve software quality.

The following summarizes our top-level requirements of a formal methods-enabling technology suitable for industrial practices as a foundation to distinguish our work from existing implementation solutions. Each numbered item becomes a research question for our work, i.e. how do we accomplish the item. An industrial-purpose formal methods-enabling tool should facilitate: 1) fully automated verification; 2) unification of languages for requirements specification; 3) analysis of static and dynamic aspects of software; 4) analysis of bounded and unbounded data-intensive systems. It should also, 5) be free to use and be open to public modifications (i.e. open-source); 6) support model analysis; 7) be domain-independent; and 8) be actively developed.

III. Related work

In the following we briefly overview work related to each of the research questions outlined in the previous section and background of the underlying technologies.

A. Overview of the Research Questions

RQ1: Supports for full automated verification - Amálio et al. [5] and others (e.g. [5], [6]) integrated formal methods with software analysis and constructions. However, the solutions provided are based on theorem proving methods. Although many theorem proving approaches claimed to be fully automatic (e.g. Vampire [7]), but they traded expressiveness for automation by relying on first-order logic notations. This problem limits the class of problems solvable by these solutions to arbitrary ones. On the other hand, higher-order logic solutions demand user-guidance and invention of lemmas in proofs search. We facilitated a fully automatic verification by adopting model checking approach, though at the expense of completeness (e.g. bounded verification in Alloy).

RQ2: A unified language for requirement specification – Cabot et al. [8] and others (e.g. [9]) adopted the Object Constraint Language [10] for requirement specification. This is similar to our work because Umple Constraint Language (Umple-CL) will be developed to allow formal specification of both dynamic and static aspects of software systems. We are proposing Umple-CL to allow analysts and designers learn one language that facilitates a forward engineering of software systems.

RQ3: Analysis of static and dynamic aspects of software systems – In [11] MDE was integrated with formal methods approaches, but, solutions offered focus on the static aspects of systems. On the other hand, in [12], [13] the dynamic aspects of software was dealt with extensively. In [9], Alloy was adopted for checking correctness of both aspects of software. Alloy is a first-order logic language; hence only trivial problems about the dynamic aspects of the system can be solved. Our work addresses both aspects of software. We adopted Alloy [14] for the analysis of static aspects and nuXmv [15] for the dynamic aspects. Thus, system analysis with our solution goes beyond trivial problems.

RQ4: Verification of bounded and unbounded data-intensive systems – Dubrovin and Junttila [16] offered a solution to analyze hierarchical state machines based on model checking. Despite the attention accorded to the work, it lacks the capability to verify data-intensive systems with unbounded domains. We realized this scalability by relying on nuXmv for the analysis of data-intensive systems.

RQ5: Open Source - Our goal is to develop an open-source technology. A large amount of solutions (e.g. [17]–[19]) have adequate successes in the industrial settings. For example, Astrée [17] gained significant attention in certifying correctness of Avionics systems [20]. However, it is not open-source; hence inaccessible for use by students and practitioners. Umple, as a formal methods-enabling tool will be open-source.

RQ6: Support for Model Analysis - Many analysis tools is program-based but not model-based. For example, JPF [21] is a "push-button" solution to analyze Java sources and object programs. Other tools in this category include [17], [22], [23]. Umple is derived from the Unified Modeling Language; hence the inputs to our tool are models as opposed to programs.

RQ7: Domain-Independent Tools – Analysis challenges cut across diverse domains (e.g. automotive [1], health [2], avionics [3], etc.). Hence, tools should be domain-independent. Static analysis tools (i.e. abstract interpretation-based) offer domain-specific solutions to achieve precision. For example, Astrée is tailored to the domain of avionics to minimize false positives and negatives. Our solution will be applicable to problems irrespective of their domains.

RQ8: Active Development – Our notion of active development refers to the last time the project received source code contributions or research publications. This is important because some projects (e.g. academic) suffer improvement often times as the project term completes or research student graduates. For example, the Symbolic Analysis Laboratory (SAL) [24] lacks active development. SAL is preferable to nuXmv because it is an open-source technology with capability to analyze unbounded types. Its lack of active development led us to choosing nuXmv as a back-end analysis engine for state machine systems (free but not open-source). Thus, we considered our work superior to tools relying on SAL for the purpose of analysis (e.g. Bandera [25]).

B. Background of the Underlying Technologies

The technologies we will discuss are Umple [26], Alloy [14] and nuXmv [15].

Umple is a model-oriented programming technology for the development of software systems. It supports the model-code duality principle by representing software models, not only as diagrams but also as text [26]. Umple allows developers to model static and dynamic views of software and automatically generates code in languages like Java, C++, Ruby, PHP from the model. Umple achieves this by providing constructs and environments to express a rich subset of Unified Modeling Language (UML) [27], such as classes, state machines, and composite structure models. It was explicitly designed to be easy to use while generating high-quality code. People used to UML diagrams can draw them in Umple (or can import them into Umple from other UML tools), but those who are used to textual coding can also use Umple.

Alloy [14] is a first-order logic language for expressing software abstractions, as well as simulating and checking requirements of software. Alloy provides syntax for expressing transitive closure, universal and existential quantifications, predicates, functions, relations, invariance, multiplicities, inheritance, and so on. With these, Alloy is suitable for representing object models, simple and complicated constraints, and operations manipulating the structures dynamically. Analysis with Alloy is fully automatic with instant feedback from its SAT-based analyzer. It adopts the *bounded model checking (BMC)* as a means of maintaining *decidability*. Thus, Alloy is *sound* but *incomplete*.

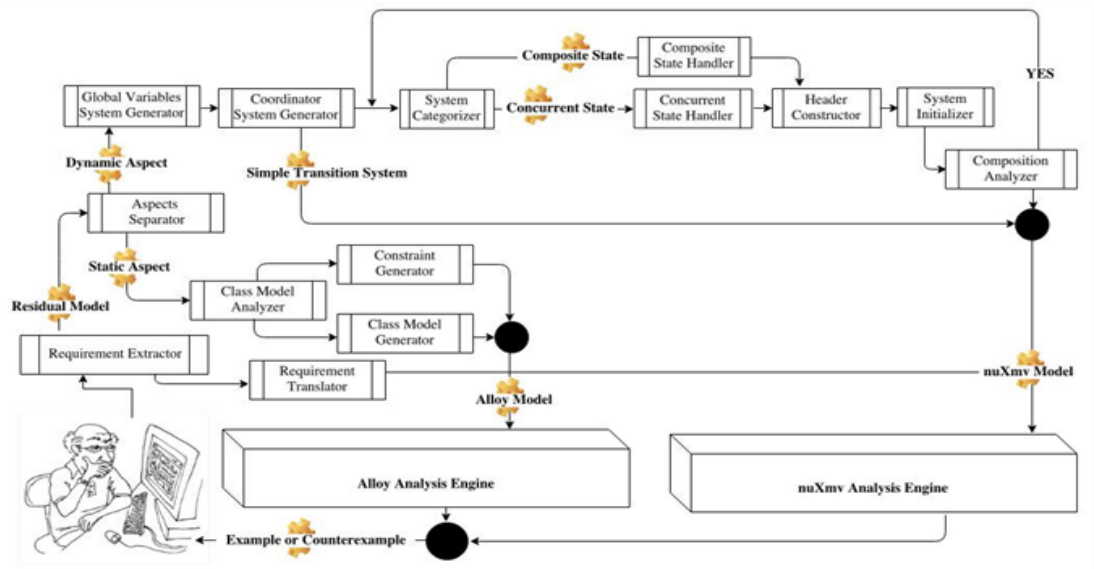

Fig. 1. Architecture for the analysis of Umple models.

Alloy analyzer is capable of discovering inconsistencies via simulations, and counterexamples by checking assertions.

nuXmv is a new symbolic model checking tool for the verification of fair finite- and infinite-state synchronous systems [15]. It extended NuSMV [28], a state-of-the-art model checker for the specification and verification of finite state systems. nuXmv adopted the basic verification techniques of NuSMV and extends its native language with *unbounded integer* and *real* types for the specification of infinite domains. For the verification of the newly supported domains, it integrates Satisfiability Modulo Theory (SMT [29]) algorithms. It has been applied in the academic and industrial contexts [30], [31]. Among model checking tools, results show that nuXmv is highly competitive [15].

IV. The Proposed Solution

Fig. 1 presents the proposed system from design to result examination phases. Readers should note that we assume the starting point is a syntactically correct static or dynamic aspect of software expressed in Umple (i.e. model and requirement). To maintain scalability of the dynamic aspects, we adopted a compositional approach to integrate hierarchical structures. We cleanly separated concerns (e.g. composite and concurrent states) and systematically (e.g. attribute access and shared variable) integrate each component of the hierarchical structure as opposed to the traditional flattening. This was proposed by [16]; but extended and implemented as an internal representation of Umple.

A. Model Preprocessing

The first phase of processing involves the extraction of requirement from the input Umple model. Aspect Separator sub-process analyzes the residual model (exclusive of requirements) to determine its kind. The residual model in this context can either be a state machine model or a class model or both. Suppose the residual model contains a state machine, the sub-process extracts the state machine and passes it to Global Variables System Generator sub-process to initiate the generation of its SMV code. On the other hand, suppose the residual model contains a class model, this is being extracted and passed to the model analyzer to initiate the generation of its Alloy code.

B. Formal Specification of State Machines

In this section, we discuss the components of the presented architecture responsible the specification of state machines. The Global Variables System Generator sub-process generates a transition system with only a variable declaration section. The purpose of the transition system is to allow declaration and accessibility of global variables (e.g. events enumeration variable, guard variable(s), etc.) as an entity from any point within the specification. The Coordinator System Generator sub-process creates a transition system (containing all sections) for the parent (or coordinating) state machine. Composite State Handler sub-process creates a new transition system for composite states. The transition system (all sections inclusive) is semantically equivalent to a state machine corresponding to the state. Its variable declaration section defines a state variable which enumerates the set of its sub-states and a ‘null’ value (to disable transition system before entry or after exit). In its assign section, the state variable is initialized to ‘null’.

The activation of a composite state is accomplished by transition leading to it. Suppose the state is a concurrent state, the specification is delegated to Concurrent State Handler sub-process. It ensures no transition system is generated for the state but a transition system is generated for each of its sub-state. For each transition system generated for the sub-states, their state variables enumerate the state name, other sub-states The activation of all sub-states of a concurrent state is accomplished by transition leading to it. Head Constructor sub-process constructs the header of every transition system created for the input state machine models. For every transition system, instances of all transition systems created (including transition system for global variables) will be passed as arguments to each transition system (except the transition system under construction). Within the transition part of every transition system with a nested state, the creation

of a nuXmv specification for the composite state and a transition whose next state is the initial state of the composite state is delegated to System Initializer sub-process. The System Categorizer sub-process analyzes the input state machine to determine if it has a composite or concurrent state or is a simple state machine. Suppose the input state machine has a composite state, the Composite State Handler sub-process is invoked; but if the state machine has a concurrent state then Concurrent State Handler sub-process is invoked. However, if the state machine is a simple state machine (i.e. none of its states are nested) then the generation terminates.

The Composition Analyzer sub-process determines termination for a state machine with a composite or a concurrent state. Suppose a composite or concurrent state has a nested child state then control is transferred to System Categorizer sub-process. In this case, the processing proceeds recursively until a termination is reached. For both simple and hierarchical transition systems generated, our engine generates a 'main' for the instantiation and execution of the resulting transition system(s); just as the 'main' program is the entry point of execution of systems developed for programming languages that inherit notions facilitated by C-programming.

C. Formal Specification of Class Models

The analysis of static class models begins with the model analyzer. This component is responsible for the analysis of models to discover various design patterns characterizing the model. These patterns are to be used by the model and constraint to produce a semantically equivalent formal specification of the discovered patterns. The model generator is tailored to associate relevant code to patterns discovered by the analyzer. Similarly, the constraint generator is tailored to associate relevant constraint for each pattern discovered. The outputs of both generators are combined to form the formal representation of the input class model.

Generally, the equivalent specification for the requirement of the model under analysis (MUA) is generated by a translator. Therefore the model generated (Alloy model and nuXmv transition system(s)) and its requirements are fed into the target analysis engines (Alloy analyzer and nuXmv engine respectively). The result of analysis is either an example or a counterexample depending on the analyst's intent or the model expressed. This will be translated to appropriate structures (e.g. class diagrams) to facilitate easy understanding by the analyst. For code generation, we adopted template and meta-modeling approaches. We defined a meta-model for the target languages and a set of templates for cases under consideration (e.g. pattern constraints, transition systems, etc.).

V. PRELIMINARY WORK

The current status of this work has been facilitated by the following research activities:

- a systematic review of existing implementations;
- the design of meta-models for the formal languages;
- an incremental development of constraints for some object-oriented patterns;
- the design of test cases for various modeling patterns; an exploration of representation schemes, particularly, the encoding of composite and concurrent state machines;
- the implementation of prototype code generators for static and dynamic aspects of software; and simulation of constraints for static aspects and behavioral properties for dynamic aspects of example systems.

Results obtained demonstrated the correctness of our code generators; adequacy of the structural constraints; weaknesses of the initial encoding and strengths of the current encoding. For example, our initial strategy enumerates all states of hierarchical system as elements of a variable as an approach to manage the state space. The limitation of this style was exposed when we applied it to concurrent systems, because a variable cannot assume more than one state in a time unit.

VI. EXPECTED CONTRIBUTIONS

This general goal requires solving various sub-problems, each of which is a distinct contribution. The expected contributions are the following:

- A scalable symbolic encoding of state machine systems that combines data-/control-intensive aspects of software systems.
- A concrete pattern-based approach to the formal specification of class models.
- Transformation tools from Umple (and hence from UML) to nuXmv and Alloy and back to Umple.
- Improvements to model analysis.
- Understanding of limits of Umple, nuXmv, and Alloy.
- Extensions to Umple.
- Use in practice – Case studies.

VII. PLAN FOR EVALUATION AND VALIDATION

To ensure correctness of our system, we will adopt the following strategies for verification purposes. These are simulations and test-driven development. To this point, we have adopted these approaches and obtained promising results for the tasks.. In the final thesis, we will explore various approaches as other practical challenges unfold.

We will develop case studies in various domains (e.g. automotive) to validate our approach. The case studies will be made of models and their respective correctness requirements. We will apply our system to generate the formal specifications of the system. Finally, we will verify the correctness of each system against its requirements by invoking our back-end analysis engines.

VIII. CURRENT STATUS

As of the writing of this paper we have selected the tools, prototyped the generation of nuXmv and Alloy from Umple, and explored a variety of examples. We faced many challenges in reaching this stage, most notably working out how best to represent Umple semantics in the selected formal languages in a modular way. Overcoming these challenges has led to some of the additional contributions of this work.

Between now and the time of the thesis submission, we expect to continue to refine our formal-methods generation capability, validate our approach on a wide variety of examples, and make some corrections to Umple as our work uncovers them.. These activities will be guided by the following timeline:

TABLE 1. TIMELINE FOR RESEARCH COMPLETION

MONTHS	ACTIVITIES
August 2015 – March 2016	Systematic refinement of formal-methods code generation capability.
April 2016 – July 2016	Validation with variety of case studies.
August 2016	Submission of thesis.

IX. ACKNOWLEDGMENT

I would like to thank Shoham Ben David for her insightful contribution in this research. Similarly, I would like to specially acknowledge the supervisory roles of Timothy Lethbridge and Stéphane Somé.

X. REFERENCES

[1] J. Schauffele, T. Zurawka, and S. Germany, *Automotive Software Engineering*. 2005.

[2] N. G. Leveson and C. S. Turner, "An Investigation of the Therac-25 Accidents," *Computer (Long. Beach. Calif).*, vol. 26, no. 7, pp. 18–41, 1993.

[3] J. Souyris and D. Delmas, "Experimental assessment of Astree on safety-critical avionics software," in *Proceedings of the 26th international conference on Computer Safety, Reliability, and Security*, 2007, pp. 479–490.

[4] P. Codd, "Top 10 Software Failures Of 2011." [Online]. Available: http://www.businesscomputingworld.co.uk/top-10-software-failures-of-2011/. [Accessed: 02-Apr-2015].

[5] W. Ahrendt, B. Beckert, and D. Bruns, "The key platform for verification and analysis of Java programs," *Verif. Softw. Theor. Tools, Exp. (VSTTE 2014)*, pp. 1–16, 2014.

[6] H. Z. H. Zhu, I. Bayley, L. S. L. Shan, and R. Amphlett, "Tool Support for Design Pattern Recognition at Model Level," in *33rd Annual IEEE International Computer Software and Applications Conference*, 2009, vol. 1, pp. 1–6.

[7] A. Riazanov and A. Voronkov, "The design and implementation of Vampire," *AI Commun. - CASC*, vol. 15, no. 2, pp. 91–110, 2002.

[8] J. Cabot, R. Clarisó, and D. Riera, "UMLtoCSP: A tool for the formal verification of UML/OCL models using constraint programming," *ASE'07 - 2007 ACM/IEEE Int. Conf. Autom. Softw. Eng.*, pp. 547–548, 2007.

[9] B. Bordbar and K. Anastasakis, "UML2ALLOY: A tool for lightweight modelling of discrete event systems.," *Guimarães, N., Isaías, P. IADIS Int. Conf. Appl. Comput. 2005*, no. 1999, pp. 209–216, 2005.

[10] Omg, "OMG Unified Modeling Language: Version 2.5," vol. 05, no. September, p. 786, 2013.

[11] M. Balaban and A. Maraee, "Finite Satisfiability of UML Class Diagrams with Constrained Class Hierarchy," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 3, pp. 24:1–24:42, 2013.

[12] K. Zurowska and J. Dingel, "Symbolic execution of UML-RT State Machines," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC '12*, 2012, p. 1292.

[13] M. Bakera, T. Margaria, C. D. Renner, and B. Steffen, "Tool-supported enhancement of diagnosis in model-driven verification," *Innov. Syst. Softw. Eng.*, vol. 5, no. 3, pp. 211–228, 2009.

[14] D. Jackson, *Software Abstractions*. Massachussets: The MIT Press, 2012.

[15] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The NUXMV Symbolic Model Checker," in *26th International Conference on Computer Aided Verification*, 2014, pp. 334–342.

[16] J. Dubrovin and T. Junttila, "Symbolic model checking of hierarchical UML state machines," in *8th International Conference on Application of Concurrency to System Design*, 2008, pp. 108–117.

[17] P. Cousot, R. Cousot, J. Feret, A. Miné, and X. Rival, "The Astrée Static Analyzer," 2015. [Online]. Available: http://www.astree.ens.fr/. [Accessed: 06-Jun-2015].

[18] J. Ivers, "Overview of ComFoRT : A Model Checking Reasoning Framework," 2004.

[19] T. Ball, V. Levin, and S. K. Rajamani, "A decade of software model checking with SLAM," *Commun. ACM*, vol. 54, pp. 68–76, 2011.

[20] O. Bouissou, E. Conquet, and P. Cousot, "Space software validation using abstract interpretation," in *Proc. of the Int. Space System Engineering Conf., Data Systems in Aerospace (DASIA 2009)*, 2009, no. 1, pp. 1–7.

[21] K. Havelund and T. Pressburger, "Model checking JAVA programs using JAVA PathFinder," *Int. J. Softw. Tools Technol. Transf.*, vol. 2, pp. 366–381, 2000.

[22] P. Chalin, J. Kiniry, G. Leavens, and E. Poll, "Beyond assertions: Advanced specification and verification with JML and ESC/Java2," *Form. methods components objects*, vol. 4111, pp. 342–363, 2006.

[23] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-based Predicate Abstraction for ANSI-C," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, 2005, vol. 3440, pp. 570–574.

[24] L. de Moura, S. Owre, and N. Shankar, "The SAL language manual," *Comput. Sci. Lab., SRI Int., Menlo …*, vol. 02, no. 650, pp. 1–39, 2003.

[25] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, R. Robby, and H. Z. H. Zheng, "Bandera: extracting finite-state models from Java source code," in *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*, 2000, pp. 439–448.

[26] O. Badreddin and T. C. Lethbridge, "A manifestation of model-code duality: facilitating the representation of state machines in the umple model-oriented programming language," 2012.

[27] J. Rambaugh, I. Jacobson, and G. Booch, *Advanced Praise for The Unified Modeling Language Reference Manual , Second Edition*, 2nd ed. Toronto: Addison-Wesley, 2004.

[28] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2 : An OpenSource Tool for Symbolic Model Checking," pp. 359–364, 2002.

[29] L. De Moura and N. Bjørner, "Satisfiability modulo theories: An appetizer," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 5902 LNCS, pp. 23–36, 2009.

[30] A. Cimatti, R. Corvino, A. Lazzaro, I. Narasamdya, T. Rizzo, M. Roveri, A. Sanseviero, and A. Tchaltsev, "Formal verification and validation of ERTMS industrial railway train spacing system," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 7358 LNCS, pp. 378–393, 2012.

[31] A. Cimatti, S. Mover, and S. Tonetta, "SMT-based scenario verification for hybrid systems," *Form. Methods Syst. Des.*, vol. 42, no. 1, pp. 46–66, 2013.