Combining Verification and Validation techniques

Erwan Bousse

erwan.bousse@irisa.fr
University of Rennes 1, IRISA, France

Abstract. As systems are more and more complex and heterogeneous, Domain Specific Languages (DSLs) can be used to model their many concerns at a high abstraction level. Early Validation and Verification (V&V) of such models is an important and difficult task. There are numerous possible V&V techniques, none being completely satisfying regarding specific requirements (behavioral coverage, scalability, required expertise, etc.) or depending on the context. The objective of my PhD work is to find ways to make an efficient coordinated use of different complementary V&V techniques. In particular, I focus on the need to handle gathered *evidence* with corresponding *property-behavior coverage*, and on the semantic gap between domain languages and V&V languages. The proposed approach include the designing of a language to encode V&V evidence, and on the use of model types to both capitalize V&V manipulations and fill the gap between DSLs and V&V languages.

Keywords: Verifcation, validation, model driven development, domain specific languages, partial evidence, model typing.

1 Introduction

In many fields, one can observe an increase of systems complexity and size. To handle such complexity, the use of Domain Specific Languages (DSL) within Model Driven Development (MDD) processes can be considered for many types of systems (e.g. planes, cars, satellites, rail systems). In order to improve the quality of such systems and to reduce safety or security risks, Verification and Validation (V&V) is of first importance. Performed as early as possible, it can avoid costly impacts of design errors or unmet requirements. More specifically in the safety-critical systems field, compliance with standards such as EN 50128 (railway applications), IEC 61508 (electronic safety related systems), or DO 178C (software-based aerospace systems) must be ensured.

With these goals in mind, the question of *how* to perform effective V&V on such complex and potentially heterogeneous models must be considered. But there is no simple answer to this question, as there are various way to verify and validate a system: testing, model checking, theorem proving to name a few. As all these techniques come from different communities, they are very different regarding how they work, their ease of use, or their outputs. Considering that none of these techniques nor tools that support them are perfect for a given

context (ie. the system of interest, its environment, its requirements), I will address during my PhD the idea of *combining* different V&V techniques together. Combining different V&V techniques can be defined as using multiple V&V techniques to analyze the specified properties of a modeled system.

In this paper, I first give some possible characteristics of V&V techniques in order to highlight the benefits on V&V techniques combinations. Based on the ideas of *partial evidence* and *property-evidence coverage* from Dwyer and Elbaum [9], I then review the challenges I identify for this problem: the integration of V&V techniques and the semantic gap between DSLs and V&V languages. Finally, I give some possible solutions through the approach I propose: the modeling of gathered evidence and reached property-coverage, and the use of model typing to conveniently bring closer DSLs and V&V languages.

The remaining sections are organized as follows. Section 2 is the detailed description of the studied problem. Section 3 presents some related work on the subject. Section 4 contains the description of the approach I propose. Finally, I conclude in Section 4.

2 Problem description

There are multiple ways to perform V&V. Testing, model checking, theorem proving, runtime monitoring or static analysis are among the best known approaches. Each of them and/or each of the tools that support them have specific characteristics, such as:

- their input languages (level of abstraction, formalized or not, domain specific or general purpose, etc.)
- the kinds of properties they can prove (safety or liveness, temporal or not, syntactic consistency, absence of null pointer dereferencing, etc.)
- coverage (compliance for one/few/all execution trace(s))
- scalability (small/big systems, complex/simple properties)
- results they produce (formal proof, "out of memory" error, positive outcome, counter example, etc.)
- the required level of expertise (from domain specific developers to formal method experts)

This results in many differences, but also different strengths and weaknesses. For instance, manual testing is a well-know approach for most software developers, but can lack exhaustiveness in terms of behavioral coverage if not done rigorously; some model checking tools do not require much expertise and explores all states of the system, but may not scale to huge systems [7]; theorem proving produces proofs that can be kept to certify the model, but needs appropriate experts. Moreover, there also are many differences between tools of the same family; for instance the NuSMV model checker [5] considers the SMV models with CTL (Computation Tree Logic) or LTL (Linear Temporal Logic) properties, whereas the UPPAAL model checker [19] can analyze timed-automata with TCTL (Timed CTL) properties.

Intuitively, one might consider that these different approaches and/or the tools that support them are *complementary* in some way. On this subject, Dwyer and Elbaum defend in [9] the idea of *unifying V&V techniques* in order to produce evidence that assert that a system meets its specified properties. More specifically, they consider the notion of *partial evidence*, based on the fact that no analysis can perfectly judge a system regarding a property. They introduce a metric called *property-behavior coverage*, which measures how much are properties proven regarding the possible behaviors of the system. For instance, an analysis may prove all specified properties on a small subset of behaviors (e.g. testing) while another may analyze all behaviors for a single property (e.g. static program analysis, with some imprecision). In between, some techniques can prove all properties for all behaviors, but only on an abstraction of the considered model (e.g. some model checkers). In a nutshell, by gathering many pieces of partial evidence from multiple and different V&V techniques, one can eventually be confident enough that a system meets its properties.

Therefore, the main question that I will tackle during my PhD is the following: how can we *combine* different V&V techniques and evidence they produce? More precisely, my work will focus on the two following aspects.

Integration of V&V techniques and evidence sharing. The main issue is the proper integration of multiple V&V techniques into a single process. In other words, we need bring together and link in a defined way multiple V&V tools within a process. Two challenges stated by Dwyer and Elbaum [9] are the need for *"standardized syntax and semantics for evidence to enable tools to interchange and process evidence"*, and the need to agree on *"frameworks for encoding property-behavior coverage"*. It seems indeed inevitable that results must be shared between tools, whether they are concrete evidence (formal proof, execution trace, etc.) or information about property-behavior coverage. For instance, a theorem prover of high level properties should be informed if some low level properties were already proven by a model checker; moreover, this should clearly appear in the formal proof that the former will provide. Besides the need to encode results, there is the broader issue of connecting V&V tools altogether around gathered evidence and to automatize their calls.

Semantic gap between domains and V&V tools. In the context of MDD processes and DSLs, a way to perform V&V is to translate domain models into models conforming to existing V&V languages (ie. input languages of existing V&V tools) [24,4]. This approach allows a reuse of existing V&V tools and prevents the tedious task of building new ones for DSLs. Figure 1 shows the most widespread ways to accomplish such capitalization. The most common solution (a) is to define direct transformations towards V&V languages, but this has at least two drawbacks: the semantic gap to fill is quite important, and a significant number of transformations are required. To cope with these issues, a second solution (b) is to rely on a *pivot language*, which decreases the semantic gap between two sets of languages and allows a capitalization of transformations towards V&V languages. Yet, this solution raises several problems. In particular, the

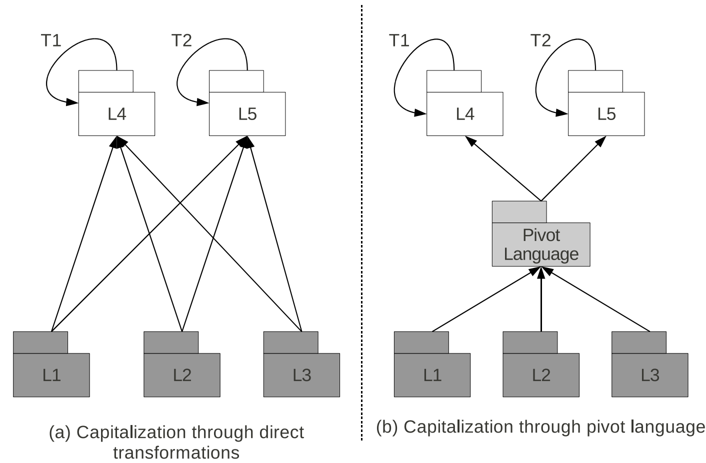

Fig. 1. Common approaches for the capitalization of model manipulations

idea of pivot language implies a certain universality of the pivot. Every concept possibly defined in existing or future languages has to be somehow included in this language, and existing work has shown the difficulty of such a task. For instance, TOPCASED has defined the pivot language Fiacre [2] between DSLs (e.g., UML, AADL, etc.) and various V&V environments (Petri nets and timed automata). Since there were too many concepts to consider within the pivot, a pivot language *family* was eventually designed, including among other synchronous and asynchronous versions. Concisely, it is difficult to implement this solution in the general case (i.e., for any set of languages).

3 Related work

Considerable effort has already been done to bring together multiple V&V approaches, although to my knowledge none is a general approach nor completely solves the subproblems identified in the previous section. Therefore, this section is not a comprehensive overview and only presents interesting work on the subject in some specific contexts.

Bhadra et al. [3] made a very rich survey of existing hybrid approaches for functional verification of hardware designs. Among other, many attempts to integrate model checking with theorem proving are described. Model checking can be used as a decision procedure for theorem proving [18], or to prove low level properties of systems to discharge a theorem prover that focuses on higher level ones [16,21]. Also, many modern theorem provers provide ways to connect with over tools; e.g. PVS provides connections with model checkers and SAT solvers [22] and Isabelle uses external tools as oracles for checking formulas during a proof search [1]. There also are approaches that combine distinct model checkers based on different algorithms (e.g. BDD or SAT), one being better for a task than

another [15]. With this survey, Bhadra highlights the feasibility and the benefits of combining V&V techniques. Besides, as each of these approaches focuses on at most two V&V tools within precise context, it also shows that a more global approach to implement these integrations would be of much interest.

There also are examples of hybrid approaches in the field of Software Engineering. Hazelhurst et al. [25] worked on the tool Palus, which relies on both dynamic and static program analyses to guide a random test generator. In a similar fashion, Ge et al. [11] developed the tool DyTa that uses static program analysis to detect potential defects, and then uses dynamic test generation to confirm or infirm these defects. The work of Filieri et al. [10] considers the use of a model checker in order to generate expressions used for runtime monitoring. However most of these approaches focus mainly on programs, while I would like to tackle more general MDD processes. Moreover, these are quite sophisticated combinations of V&V techniques which may be difficult to consider in a more general approach.

The Hi Lite project [8] aims to bring new ways to use formal methods within real size projects. In particular, they defend the idea that formal methods won't ever bring total proofs of correctness of program, but should nonetheless have a main role in V&V effort. The tool GNATProve [12] is being developed by AdaCore in the scope of this project, and aims at combining multiple theorem provers in order to provide fully automatic verification of a verifiable subset of Ada. Following the same idea, the recent work of Comar et al. [6] suggests the use of formal methods on verifiable pieces of software, while testing the other parts. The project is influenced by with the new DO-178C standard for aircraft software [20], which henceforth allows the use of formal methods to discharge part of the mandatory testing. Yet, this project focuses on low-level programming languages (C and Ada), while my work will consider MDD processes and DSLs.

4 Proposed approach

In this section, I present the high-level solutions that I envision to cope with the problems presented in Section 2.

4.1 Modeling V&V evidence and property-behavior coverage

As explained in Section 2, an approach to integrate multiple V&V techniques within a MDD process needs proper encoding of both *evidence* produced and *property-behavior coverage* reached. Many actors and concepts must be considered for such a framework, such as:

- the forms of evidence (formal proof, model checker positive result, counter example, out of memory error, ongoing proof with subgoals, etc.);
- the tools used for the analyses;
- the specification languages (logics, natural language, code assertion);
- the behaviors for which properties are proven (all states of the system, a specific trace, states that meet an assumption, etc.);

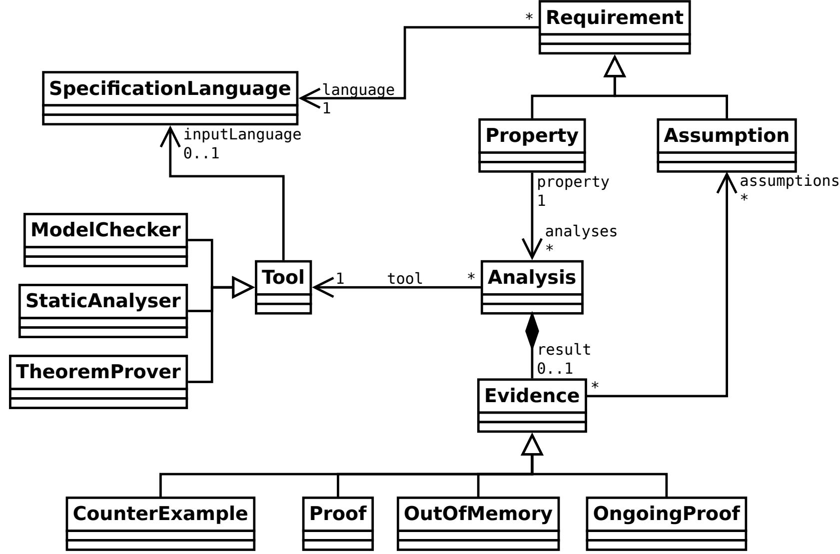

Fig. 2. Intuitive framework for V&V techniques

Figure 2 is a very approximate draft of how such a framework could look like. A Requirement can be a logic formula, a sentence in natural language, or a Büchi automaton (that can encode a LTL formula) depending on its SpecificationLanguage. A requirement is either a Property to prove or an Assumption concerning the system and its environment. For each property, multiple analyses (Analysis metaclass) can be performed – each by a Tool – and provide Evidence that can be of many form. More importantly, to model what behaviors are covered by some evidence, we choose here to consider a member `assumptions` for Evidence that states under which conditions the property is valid; that should be refined in the expected PhD work.

Yet, there are even more aspects to consider. Figure 2 does not detail the structure of the notion of evidence; this task appears difficult regarding the various forms of evidence that exist. A similar problem concern properties, whose specification language can drastically change their structure. For this reason, and for better usability, it is very likely that logical representations will be favored. Modeling testing is also an issue, since other notions must be considered (e.g. execution platforms). Last but not least, how could this language be concretely used to integrate V&V techniques? Work of Guttman et al. [13] around interface logics may prove of interest: they suggest the combination of existing logics – then considered as sublogics of a bigger one – in order to combine theorem provers. Part of my work will consist in trying to elaborate this framework and to experiment with it within a process involving combined V&V techniques.

4.2 Model typing for V&V

Part of the problem described in Section 2 concerns the semantic gap between DSLs and V&V input languages. As this question has not been perfectly answered yet in the general case for a single DSL and a single V&V language, it gets even more complex when trying to combine *multiple* V&V techniques.

Work on the notion of *model typing* [14,23] is of much interest to solve this problem. The idea behind model typing is to consider subtyping relations between metamodels, with substituability in mind. For example, if a metamodel B is a subtype of a metamodel A, then a model conforming to B can be manipulated as if it was an instance of A.

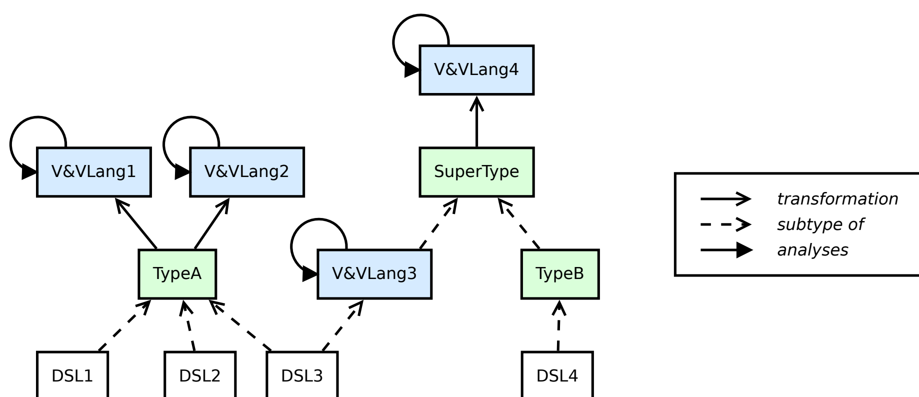

Fig. 3. Capitalization of V&V analyses through model typing and a type hierarchy

In [17], Jézéquel et al. give many thoughts on the potential of model typing regarding capitalization of model manipulations. In particular, the idea of a type hierarchy is very attractive: by defining a partially ordered set of languages, one can reuse manipulations in an elegant way. Figure 3 shows how can this idea be applied to V&V, in the context of multiple DSLs and multiple V&V languages. Multiple cases must be considered. An intermediate type can be defined to factorize transformations towards similar V&V languages – see DSL1, 2 and 3 subtypes of TypeA, which can be transformed into V&VLang1 and 2. Another possibility is to type DSLs with V&V language in order to directly feed them to corresponding V&V tools – see DSL3 and V&VLang3. Finally, a higher type can encompass multiple other types – see DSL3 and 4 which are indirectly of type SuperType.

For the studied problem, the benefit of model typing is threefold. First, it is a way to fill the semantic gap between DSLs and V&V languages by relying on subtyping relations and underlying adaptations, which allows better reuse of manipulations and reduce the number of transformations required. Second, model typing can be a way to formalize in a very intuitive way what possibilities

of V&V we have for considered DSLs. Third, one can statically reason on the types of the studied models in order to make decisions during V&V, which meets into the aforementionned V&V techniques integration problem. As the main goal is to combine V&V techniques, to model and to be able to use such information could be a key. Nonetheless, many questions remain concerning *how* exactly to use model types for V&V. Should we consider types influenced by transient models of V&V techniques (e.g. transition systems for model checkers algorithms), or should we stay closer to input languages of V&V tools (e.g. SMV or Promela model checker languages)? Part of my work will consist in answering these question and to fully understand the potential of model typing for V&V.

5 Conclusion and further work

In this paper, I present the wide problem of combining V&V techniques and the approach I consider to tackle it. V&V techniques all have strengths and weaknesses that justify the idea of using them together. This is confirmed by the many existing hybrid V&V approaches, which not only prove that this is possible, but also show that a more global way of thinking this wide problem would bring a lot to future integrations. In particular, I plan to tackle two aspects of the problem. On the one hand, the integration of multiple techniques itself, which includes the need to manipulate and exchange gathered evidence between V&V tools. On the other hand, the semantic gap between DSLs and V&V languages, which gets even more complex in the case of multiple languages on both sides of the required translations. The proposed approach is the designing of a language to model evidence and property-behavior coverage, and the use of model typing to bridge the semantic gap and capitalize V&V manipulations.

Many questions remain unanswered, whether concerning the criteria to consider for an evidence language, or the integration of tools together using such language, or how exactly could model types be used for V&V. Another challenge that must be considered is the process of deciding what V&V techniques should be used to prove a specific property of a system. This may involve a method to analyze and categorize specified properties. Last but not least, Bhadra et al. [3] emphasize on the importance of *soundness* of integration of V&V techniques. One must be cautious not to undermine trust in V&V tools by implementing wrong connections between them.

The next step of my work will consist in experimenting with a use case and a pair of V&V approaches, which will most probably be random testing and model checking. My objectives are to define all concepts that this case would require within a framework for V&V evidence, and also to define model types that could be used within the process.

References

1. David Basin and Stefan Friedrich. Combining WS1S and HOL. *Frontiers of Combining Systems*, 2:39–56, 1998.
2. B Berthomieu, JP Bodeveix, and P Farail. Fiacre: an intermediate language for model verification in the TOPCASED environment. *ERTS 2008*, pages 1–8, 2008.
3. Jayanta Bhadra, MS Abadir, LC Wang, and Sandip Ray. A survey of hybrid techniques for functional verification. *IEEE Design and Test of Computers*, 24.2:112–122, 2007.
4. Erwan Bousse, David Mentré, Benoît Combemale, Benoît Baudry, and Takaya Katsuragi. Aligning SysML with the B method to provide V&V for systems engineering. *Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa '12)*, 1:11–16, 2012.
5. Alessandro Cimatti, Edmund Clarke, and E Giunchiglia. Nusmv 2: An opensource tool for symbolic model checking. *Computer Aided Verification*, 2404:359–364, 2002.
6. Cyrille Comar, Johannes Kanig, and Yannick Moy. Integrating formal program verification with testing. *Proceedings of the Embedded Real Time Software and Systems Conference, ERTS*, 2:1–11, 2012.
7. S. Demri, F. Laroussinie, and Ph. Schnoebelen. A parametric analysis of the state-explosion problem in model checking. *Journal of Computer and System Sciences*, 72(4):547–575, June 2006.
8. Robert Dewar. A Pragmatic View of Formal Methods: the Hi-Lite Project. In Chris Dale and Tom Anderson, editors, *Advances in Systems Safety*, pages 233–248. Springer London, 2011.
9. Matthew B. Dwyer and Sebastian Elbaum. Unifying verification and validation techniques: relating behavior and properties through partial evidence. *Proceedings of the FSE/SDP workshop on Future of software engineering research (FoSER '10)*, pages 93–97, 2010.
10. Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. Run-time efficient probabilistic model checking. In *Proceedings of the 33rd international conference on software engineering*, pages 341–350. ACM, 2011.
11. Xi Ge, Kunal Taneja, Tao Xie, and Nikolai Tillmann. DyTa: dynamic symbolic execution guided with static verification results. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, pages 992–994. IEEE, 2011.
12. J Guitton, Johannes Kanig, and Yannick Moy. Why Hi-Lite Ada? *Proceedings of Boogie 2011, the 1st International Workshop on Intermediate Language Verification*, 2011.
13. Joshua D. Guttman. A Proposed Interface Logic for Verification Environments. Technical report, 1991.
14. Clément Guy, Benoît Combemale, and Steven Derrien. On model subtyping. *Modelling Foundations and Applications*, pages 400–415, 2012.
15. Scott Hazelhurst, Gila Kamhi, Osnat Weissberg, and Limor Fix. A hybrid verification approach: Getting deep into the design. In *Design Automation Conference, 2002. Proceedings. 39th*, pages 111–116. IEEE, 2002.
16. Scott Hazelhurst and C-JH Seger. A simple theorem prover based on symbolic trajectory evaluation and BDD's. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 14(4):413–422, 1995.

17. Jean-Marc Jézéquel, Benoit Combemale, Steven Derrien, Clément Guy, and Sanjay Rajopadhye. Bridging the chasm between MDE and the world of compilation. *Software & Systems Modeling*, 11(4):581–597, August 2012.
18. Jeffrey J Joyce and Carl-Johan H Seger. Linking BDD-based symbolic evaluation to interactive theorem-proving. In *Proceedings of the 30th international Design Automation Conference*, pages 469–474. ACM, 1993.
19. KG Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, pages 134–152, 1997.
20. Yannick Moy, Emmanuel Ledinot, H Delseny, Virginie Wiels, and Benjamin Monate. Testing or Formal Verification: DO-178C Alternatives and Industrial Experience. *IEEE Software*, pages 1–18, 2013.
21. C-JH Seger, Robert B Jones, John W O'Leary, Tom Melham, Mark D Aagaard, Clark Barrett, and Don Syme. An industrially effective environment for formal hardware verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(9):1381–1405, 2005.
22. Natarajan Shankar. Using decision procedures with a higher-order logic. In *Theorem proving in higher order logics*, pages 5–26. Springer, 2001.
23. Jim Steel and Jean-Marc Jézéquel. On model typing. *Software & Systems Modeling*, 6(4):401–413, January 2007.
24. Willem Visser, Matthew B. Dwyer, and Michael Whalen. The hidden models of model checking. *Software & Systems Modeling*, pages 541–555, August 2012.
25. Sai Zhang. Palus: a hybrid automated test generation tool for java. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, pages 1182–1184. IEEE, 2011.