From Model Driven Engineering to Verification Driven Engineering

Fabrice Kordon[1], Jérôme Hugues[2], and Xavier Renault[1]

[1] Université Pierre & Marie Curie, Laboratoire d'Informatique de Paris 6/MoVe
4, place Jussieu, F-75252 Paris CEDEX 05, France
xavier.renault@lip6.fr,fabrice.kordon@lip6.fr
[2] GET-Télécom Paris – LTCI-UMR 5141 CNRS
46, rue Barrault, F-75634 Paris CEDEX 13, France
jerome.hugues@enst.fr

Abstract. The definition and construction of complex computer-based systems require not just software engineering knowledge, but also many other domain-specific techniques to ensure many system's functional and non-functional properties. Hence, there is a trend to move away from programming languages to models on which one can reason: model-driven engineering. Yet, this remains a complex task: one need to master many techniques. In this paper, we claim that MDE is incomplete: it is "just" an implementation framework to support advanced model-based techniques, verification of systems non-functional properties, code generation, etc. There is a conceptual gap to fill to know "what" to do with models. We propose to switch from MDE to VDE: Verification-Driven Engineering, so that the user knows how to model a system to analyze it. We sum up existing techniques and their relevant application domains.

1 Introduction

Industry-critical applications are facing multiple dimensions challenges: increasing interaction patterns from traditional one-to-one to large scale peer-to-peer interaction; support for multiple level of assurance like security, reliability, timeliness. A synthesis of these challenges is faced by ubiquitous-like systems, which increase complexity due to their massively parallel execution and the variety of participants (infrastructure, service provider, user peers, etc). The notion of quality is therefore hard to define and must reflect both the notion of service provided, and corresponding level of support to meet user expectations in terms of cost and criticality such as mission or life-critical.

There is a trend to extend classical development methods to reduce such complexity. Model Driven Engineering (MDE) proposes a first step to reach that goal by using models (specifications) at every stage of the software life cycle [54]. This approach is also called MDD for Model Driven Development [55]. Development becomes "model centric". Eventhough this idea seems appealing, some issues are raised.

In a MDE setting, the engineer first models a system, implements and validates it. Testing distributed programs cannot be done easily due to the interleaving of several instruction flows. Some more adequate abstraction is needed to perform reasoning on the system and deduce undesired behavior or situations. As such, exploitation of models in a MDE approach means nothing if the underlying techniques to process the model are not efficient enough, or even non-existent.

It is now well accepted that traditional simulation approaches are not satisfactory when applied to models: it is impossible to *compute* properties or *systematically* detect unexpected situations. There is a need for formal methods to reason on specifications.

However, using formal methods is more difficult that one could expect [29], even if maturity in that domain grows (industrial tools are now available such as Atelier-B [5] or SCADE [26]). There is still a methodological issue for engineers who are not specialists of formal methods but want to use them to validate some aspects of their system. The key challenge is to know how to select techniques, to determine when these techniques are relevant and what benefits we can expect from the existing tools.

Hence, we propose to move from MDE (Model Driven Engineering) to VDE (Verification Driven Engineering). In this context, we claim that one need to consider Model-Driven Engineering as a generic framework in which verification plays a specific role at several points in the development, from early validation phase up to in-depth analysis, with benefits to quality of system to certification.

One key guideline is to provide verification facilities to system designers. Since there is no “silver bullet”, we must also provide enough information to help picking up the correct technique and tools. Hence, the formal method community must provide a classification of verification techniques, and a methodological framework so that designer can select the appropriate techniques to verify model. Finally, one must find a way to ease the use of these appropriate notations (e.g. automata, lemmas, etc.), probably by using some dedicated language(s).

The objective of this paper is first to sum up existing formal verification techniques, discuss their trade-off and see how and when they can meet engineering needs. Section 2 presents existing elements in MDE to be considered for VDE and section 3 proposes our vision of VDE.

2 Building blocks from MDE

In this section, we list existing methods and processes one can apply to build complex systems. Through MDE, one may process its model and analyze it, generate code on testbed or final hardware. Still, the question of building a processable model remains.

2.1 From Models to Model Driven Engineering

The use of models is a typical step in many engineering domains, e.g. civilian engineering use models for building bridges. Surprisingly, it expanded through the software domain only recently through OMG’s UML. It becomes of particular benefits for complex software because it helps to understand a complete problem and its potential solutions through different levels of abstraction.

As authors in [55] advocate: *Model-driven development methods were devised to take advantage of this opportunity, and the accompanying technologies have matured to the point where they are generally useful. A key characteristic of these methods is their fundamental reliance on automation and the benefits that it brings. However, as* *with all new technologies, MDD's success relies on carefully introducing it into the existing technological and social mix.*

The base concept of MDE is the model itself, built upon a meta-model which defines guidelines and constraints on valid models: allowed components, composition of components and related semantics checks. This view expands from the compiler-vision of programs based on a Backus-Naur formalism.

A model is nothing but a set of well-formed entities. To process the model and perform verification, one need to extract information required to perform a specific analysis. MDE, as a process, provides methods and tools to automate this analysis. It relies on implementation artifact (MOF and QVT frameworks, XML representation) to ease interoperability between tools, and to support easy construction of tools in a uniform framework like Eclipse. Yet, MDE is an implementation framework for model-based tools: one need to reflect on concepts conveyed by models to build analysis tools on top of this framework.

2.2 Formal methods and other analysis techniques

Mathematical analysis are interesting as they allow one to reason on a model based on formal grounds. Through different theories (sets, automata, stochastic, ...), engineers have access to a large panel of methods. In this section, we list some of them.

Algebraic approaches such as Z [1] or B [3] allow to describe a system using axioms and then, prove properties of this specification as a theorem to be demonstrated from these axioms. These methods allow one to check for the consistency of interfaces through a complete type checking mechanism, or even to go further and prove theorems (lemmas, invariants) on a set of interface.

These are of particular interest because the proof is parametric and abstract ; for instance a property can hold for a number of entities taken in the natural range. However, theorem provers that help elaborating the proof are difficult to use and still require highly skilled and experienced engineers.

Model checking [14] is the exhaustive investigation of a system's state space. A designer express a property to be tested on a model, using a logic formula expressing a possible behavior of the system. This formula is compared with all the paths in the system's state space. If there is a path that does not satisfy the property, then the property does not hold and the returned path exhibits a counter-example to the property.

The main advantage of this technique is that it is now fully automated. Yet, results are obtained for a particular set of resources (e.g. N threads), and can be generalized. Besides, it is theoretically limited by the combinatorial explosion and can mainly address finite systems. However, recent techniques based on so called symbolic techniques[3] allow to scale up to more complex systems. More recent studies also investigate model checking of infinite-state systems [45]. Other extensions contemplate the verification of time-related or probabilistic properties on a model.

[3] The word *symbolic* is associated with two different techniques. The first one is based on state space encoding by means of decision diagrams and was introduced in [10]. The second one relies on set-based representations of states having similar structures and was introduced in [12].

Analytical techniques defines a set of formulae that can be applied on a well-formed model. Typical example is the Rate Monotonic Analysis [43] that provides such techniques. These techniques are defined by a set of preconditions that a model must match for being amenable to analysis, and a set of computation steps to compute a metric on a model and conclude. Yet, these frameworks are limited to computable results. In this context, in-depth scheduling analysis shows complexity issues that cannot be solved.

Simulation-based techniques proposes methods to compute an estimation of some properties of a system, like the Monte-Carlo method. This technique is required when no analytical techniques can easily be derived from a system because of the many interfering factors, e.g. peers in an ad hoc wireless network under the influence of electromagnetic perturbation. In this setting, only simulation can help gaining an estimate on the achievable bandwidth. Yet, simulation-based, just like model checking suffers from the combinatorial explosion problem. Furthermore, the parameters required for the simulation might be complex. The designer need a simple way to express these parameters in a way close to the mathematical model of the underlying phenomenon.

So, if formal verification techniques are getting more mature, there is no silver bullet since no technique can be used easily on any type of problems [40]. One technique can be useful at one step of the software life cycle and irrelevant at the next . It is necessary to use the right approach at the right stage of software design and development.

2.3 Related difficulties

The use of models is delicate. Engineers must consider models quality (appropriateness) [55], language limitations [54] or methodological aspects [38]. Appropriateness of models is crucial if engineers want to reason on them. Languages involved in the design and development process must be able to capture basics required for such reasoning. Basics range from static considerations (such as the composition of interfaces, the compatibility of Quality of Service policies), to dynamic one (ensure liveness of a model up to safety properties). This is difficult to achieve in a unified way (in the meaning of UML) because the language becomes too complex to use and generalization is usually against precision that is required in industrial critical applications.

Methodological aspects are also often underestimated. The way models and associated languages are used is very important to ensure that an analysis can be performed. Concepts and details must be considered in an appropriate way. This is even more important when formal methods are involved since a detail may ruins all the effort and make the proof or the verification false because some hypothesis on the configuration have been forgotten. Furthermore, the application of formal methods may face implementation limits through the so-called “state-explosion” problem, or the inability to compute some metrics (reliability, schedulability).

It is important to clearly state what can be achieved by each family of formal methods and to know when to use them. Then, one can states that they can provide an appropriate answer to expected properties.

2.4 Towards a better use of formal methods

MDE defines a methodological framework to elaborate models, whereas formal methods exploit some information and derive some statements for a system. The key challenge is to orchestrate requirements for an easy modeling framework dedicated to the designer, and the capability to apply formal methods in an efficient and consistent way. Therefore, we propose the following requirements as a baseline to define a consistent VDE framework that integrate MDE and verification.

$R1$ A modeling notation that allows the designer to capture the multiple dimension of his system: interfaces, functionnal, non-functionnal and behavioral properties. This requires a notation that is non-ambiguous. Standards like AADL or UML and its profiles like MARTE define such framework.

$R2$ A mapping between some models elements and a mathematical framework. This requires the modeling notation to have enough semantics, properties or expression power to derive such mapping. For instance, core UML does not support scheduling entities, whereas MARTE does.

$R3$ Eventhough semantics is present, one need to focus on the expression of complex interaction patterns to be analyzed, like ad hoc networks, consensus, ...

$R2$ and $R3$ requires the intervention of the formal method community in order to guide the engineer in its modeling work. This can be a set of guidelines, wizards or specific front-end to indicate what are the relevant information to be provided.

$R4$ If multiple analyzes are required, it is important to make sure the different modeling artifacts are consistent and reflect the same model.

$R5$ An automated process should occur, to derive the engineer's model onto a model suitable for the analysis technique. Such process can be defined through the notion of "model-bus" to exchange models.

We point out that most of these requirements where already present when the UML-SPT profile was designed. Yet, the integration of tools is inefficient and support a limited set of analysis, mostly performance analysis.

3 Verification Driven Engineering

Sections 2 listed requirements for integrated verification in a MDE framework. This section defines our vision of an extended use of MDE that puts emphasis on the exploitation of models to verify and validate properties. We call it *VDE* for *"Verification Driven Engineering"*.

If modeling the system is important, engineers often forget that a model has properties that must be defined as soon as possible. The testing research field states that tests must be elaborated jointly with specifications. This is the same for modeling as it is suggested in the B approach with so called "proof obligation" [3, 23].

However, there are several types of property that should be elaborated at various levels of the software life cycle. We here propose a classification of such properties.

3.1 Classification of properties

There are three types of useful properties when designing a system: *1) Structural properties*, *2) Qualitative properties* and *3) Quantitative properties*.

Structural properties are the ones related to the structure of the system :

– connection and consistency between interfaces of system's components,
– invariants to be maintained in the system,
– fault-tree analysis (dependencies between system's components when one fails).

Most of these properties should be established at an early stage in the design process and at a coarse grained level. They can be refined later or enriched with some smaller coarse grained, when design is being detailed.

Qualitative properties deal with the behavior of the system e.g. schedulability, liveness, causality and deadlock detection.

To ensure such properties, the behavior of the system must be defined. They are usually described later in the software process, when information are known about components behavior. If specifications moves to programming early (e.g from UML class diagram directly to implementation), these properties are not set up since it is more difficult to elaborate them on programming language.

However, some recent work try to propose solutions to behavioral analysis of programs from their source code [33]. Some tools are already operational: Feaver [31] and then Modex [34]. They are able to analyze C-ansi code and perform model checking using SPIN [32]. QUASAR [27] is able to generate a communication model using Petri Nets from an Ada Program to check for communication problems (e.g. deadlocks).

Quantitative properties are used to evaluate performances of the system or to evaluate its behavior considering characteristics such as probability of actions to occur when non-determinism occurs, or the time execution time.

To set up such properties, even ore information is required such as an estimation of execution time (for time analysis).

3.2 Relations between properties, techniques and tools

Table 1 links properties to verification techniques and list well-known related tools.

We selected a set of well known verification techniques. *Simulation* is not formal but remains widely used, at least as a first approach to analyze a new system. *Semantic Analysis* is analyzing source code to make sure it does not violate some elementary semantic checks (e.g. arithmetic on integers) or more advanced one (concurrent access on variables like in the Esterel synchronous language). Different *type checking* techniques rely on calculi and are now embedded in typical programming languages like Ada, Eiffel or CAML. Other techniques like theorem proving and model checking have already been presented in section 2.2.

There are several categories of model checking that are differentiated by their combinatorial explosion: there now exist efficient techniques like symbolic representation

Frameworks	Interface consistency	System invariants	Fault-Tree Analysis	Schedulability	Liveness	Causality/deadlocks	Performance analysis	Available tools
Simulation				×				Cheddar [56], CPNTOOLS [19], Rhapsody [58], Renew [20], SCADE [26], Simulink [46]
Semantic Analysis	×			×			×	Cheddar [56], MAST [21], SPARK [51], TRAIAN [62]
Type checking	×							EiffelStudio [25], FuZZ [44], Z/EVES [48]
Theorem proving	×	×	×		×	×		Atelier B [5], Coq [16], Z/EVES [48], PVS [57]
Model Checking...	×	×		×	×	×		CHARON [59], CPN-AMI [42], FAST [41], SMV [47], SPIN [32], SCADE [26], SPOT [24]
...timed							×	CADP [61], Kronos [22], TINA [8], UPPAAL [60]
...stochastic							×	GreatSPN [30], PRISM [52], QPME [50]

Table 1. Relations between verification needs, techniques suitable to address these needs, and some related available tools.

of states based on the computation of symmetries in the system [13] or symbolic encoding of states by means of decision diagrams [10, 17, 18] that allow to cope with state space explosion. The use of symmetries can still be applied with some success for stochastic systems (like in GreatSPN) as well as some decision diagram encoding (like in PRISM). However, none of these techniques can be applied to timed analysis for which analysis limitations are reached faster.

Table 1 illustrates that some properties may be evaluated using more than one techniques. It is up to engineers to select the most appropriate one. It is important to point out that all the referenced tools may rely on different notations. For instance, the tools we refer in Table 1 for model checking with time relies on timed automata (CADP, Kronos, UPPAAL) or times Petri nets (TINA). The choice may be delicate since techniques and tools may have week and strong parts that are not the same.

3.3 Formal methods, drawbacks

There are numerous success stories in the use of formal methods in various domains. This concerns numerous formal verification approaches like general Model Checking [15], Model Checking from programs [11], Petri Net based techniques applied to telecommunication systems [9] or algebraic methods (B) applied on the MÉTEOR subway line [6]. However, the underlying techniques are not easy to operate.

First, as Table 1 illustrates, a given type of property can be verified using several techniques, and multiple tools. Each technique or tool has its advantages and drawbacks. To be efficient, engineers must select the appropriate technique and tool for his problem.

It remains an open problem since the skills needed to address this problem require experience. This is why formal verification is costly.

Second, there is a consistency problem between: *(1)* the specification, *(2)* its mapping to formal specifications (required for verification) and *(3)* its implementation. Hence, one must ensure that what is verified is what is implemented. Usually, *(1)* is a high-level (standardized) specification that is easier to handle than a formal notation. So far, there are several approaches to tackle this problem:

• Using transformation engines from MDE to perform in a rigorous way the transformation from *(1)* to *(2)* and code generation from *(1)* to *(3)*. One need to prove that the transformations are correct. See [37] for preliminary works in the context of CCM.

• Perform "extreme-programming like approach" and consider that *(1)*, *(2)* and *(3)* are the programming language [33].

• Use the formal notation as *(1)* and perform code generation from this notation [6].

• Use a pivot notation associated to *(1)* that provides a concrete semantics and translations from this pivot notation to *(2)* and *(3)* like in the MORSE project [28].

In all cases, there is an entry point notation that acts as a pivot notation which relates several types of specifications (e.g. semi-formal, formal, implementation).

Third, when does a given property should be verified during software development ? We sketch a proposal in section 3.4, based on our experience to link the verification of a given property to a step in the software life cycle.

3.4 The VDE Design process

In this section, we explicit the way VDE can be applied in a software development process. We propose an helicoidal life cycle inspired from the prototyping based approach presented in [39]. This life-cycle is illustrated in figure 1. Each loop corresponds to one refinement of the system as follows:

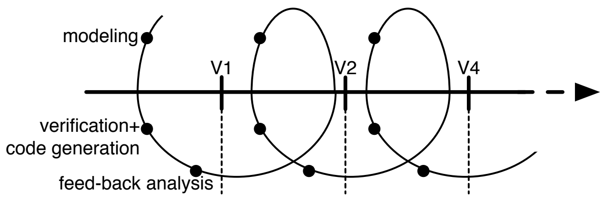

Fig. 1. The VDE helicoidal life cycle.

1. Developers must first build (or refine) a model.
2. Then, they perform some verification operations using some of the techniques and tools mentioned in table 1. To enable formal techniques, the model should be transformed into a formal specification. Otherwise, informal approaches such as simulation are also acceptable if the model remains executable.

3. If verification results are satisfactory, then the system can be generated and additional analysis can be performed (tests in the execution environment of the implemented properties). To reduce implementation costs, code generators are required.
4. Finally, analysis of collected date (from verification and execution) is stored for feed back. From this feed back, issues for the next refinement (loop) are deduced.

Towards application with AADL In [36], we evaluate potential impacts of VDE on the development of High-Integrity systems.

We contemplated the use of AADL as a mean to describe a systems on which analysis techniques can be applied. In fact, AADL is rich enough to express multiple aspects of a real-time system in a concise way.

Numerous efforts are currently done to help full analysis with some of the tools we listed: schedulability analysis with Cheddar [56]; model checking of behavioral properties with Petri Nets and CPN-AMI [42], or CHARON [59]; dimensioning analysis with OSATE [2], etc. We think this demonstrates VDE is a feasible concept.

3.5 Open issues

Of course, current practice and tools do not yet allow the full picture depicted in section 3.4. Several problems remain to have VDE fully operational. So far, there mainly are two open issues.

First, it is important to have a consistent set of information in the input specification, from which one can derive formal specifications, in a mathematical meaning

UML is a typical example: numerous works propose to derive formal specifications from specific diagrams such as state-charts like [7] or sequence-charts like [4] but none proposes simultaneous analysis from several UML diagrams. This is because connection between diagram is not formally defined. Additional interpretation must be performed (possibly by means of UML profiling like in UML-MARTE [49]). Compared to UML, AADL [53] is better to derive formal specification because all features are expressed in one single language: interface and non-functional properties are better defined and thus more exploitable for verification purpose. For instance, connections with SCADE have been improved [35]. So, we think the elaboration of a pivot notation with links to verification and code generation is mandatory.

Second, verification techniques and their related implementation are difficult to select. One tool may complete analysis on given model and be enable to cope with another one. So far, deep knowledge of the involved techniques are required. This problem is more difficult to solve, but analysis and design frameworks are still studied in the context of large projects like IST-ASSERT[4] or AVSI[5].

4 Conclusion

Bringing verification techniques to engineer is now recommended to ensure more confidence in safety critical systems. Model-Driven Engineering emerged as an efficient

[4] http://www.assert-project.net
[5] https://avsi-tees.tamu.edu/

way to reason about systems. However, MDE usually focus on the "how-to-model" rather than the "what-to-model". It is therefore difficult to know whether a system is suitable for analysis. A model that cannot be processed is useless for engineers, except for documentation purposes.

This paper proposes to reflect on concepts conveyed by models, and on existing formal methods and analysis techniques to draw a landscape of available tools. Therefore, one may move from MDE to VDE: verification driven engineering. In this context, the user would know exactly what are the facets of this system relevant for a family of analysis (e.g. schedulability), and what are the tools available to perform it.

We proposed a list of techniques and associated tools, based on a comprehensive state of art. So far, there is no silver-bullet: one need to combine multiple analysis; but also for a given analysis technique, one may need to pick the appropriate tools for interoperability, performance or its supported features.

From this complex landscape, we note there is a trend towards the integration of all these techniques around modeling notations like AADL or MARTE. We note these two notations provide strong support for the embedded systems domain.

Besides, one need to reflect on the exact goal of software engineering. Efficient application of verification techniques must be set up in a methodological approach. To do so, we propose an helicoidal cycle and advocate for its iterative nature.

From these considerations, one may provide advanced modeling tools, in which a "wizard" would guide the engineers to build its system and validate it. Defining such a process, and associated tools remain a key challenge for both the academic and industrial communities.

References

1. ISO/IEC 13568. Z formal specification notation — syntax, type system and semantics, 2002.
2. SEI AADL. Osate: An extensible source aadl tool environment. Technical report, SEI, 2004.
3. J-R. Abrial. *The B book - Assigning Programs to meanings*. Cambridge Univ. Press, 1996.
4. R. Alur, G. Holzmann, and D. Peled. An analyser for mesage sequence charts. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1055 of *LNCS*, pages 35–48. Springer, 1996.
5. Atelier B. Atelier B, the industrial tool to efficiently deploy the B Method, http://www.atelierb.eu/index_en.html, 2008.
6. P. Behm, P. Benoit, A. Faivre, and JM. Meynadier. Météor: A successful application of b in a large project. In *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems*, volume 1708 of *LNCS*, pages 369–387. Springer, September 1999.
7. S. Bernardi, S. Donatelli, and J. Merseguer. From UML sequence diagrams and statecharts to analysable petrinet models. In *Workshop on Software and Performance*, pages 35–45, 2002.
8. B. berthomieu and F. Vernadat. The TINA home page, http://www.laas.fr/tina/, 2008.
9. J. Billington, M. Diaz, and G. Rozenberg, editors. *Application of Petri Nets to Communication Networks, Advances in Petri Nets*, volume 1605 of *LNCS*. Springer, 1999.
10. J.R. Burch, E.M. Clarke, and K.L. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computation (Special issue from LICS90)*, 98(2):153–181, 1992.
11. S. Chandra, P. Godefroid, and C. Palm. Software model checking in practice: an industrial case study. In *Proceedings of the 22rd International Conference on Software Engineering, (ICSE 2002)*, pages 431–441, May 2002.

12. G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. On well-formed coloured nets and their symbolic reachability graph. In K. Jensen and G. Rozenberg, editors, *Procedings of the 11th International Conference on Application and Theory of Petri Nets (ICATPN'90). Reprinted in High-Level Petri Nets, Theory and Application.* Springer-Verlag, 1991.
13. G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. A symbolic reachability graph for coloured Petri nets. *Theoretical Computer Science*, 176(1–2):39–65, 1997.
14. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
15. E. Clarke and J. Wing. Tools and partial analysis. *ACM Comput. Surv.*, 28(4es):116, 1996.
16. CoQ Project at INRIA. The Coq proof assistant, http://coq.inria.fr/coq-eng.html.
17. J.-M. Couvreur, E. Encrenaz, E. Paviot-Adet, D. Poitrenaud, and P.-A. Wacrenier. Data decision diagrams for Petri net analysis. In *Proc. of ICATPN'2002*, volume 2360 of *LNCS*, pages 101–120. Springer Verlag, June 2002.
18. J-M. Couvreur and Y. Thierry-Mieg. Hierarchical Decision Diagrams to Exploit Model Structure. In *Formal Techniques for Networked and Distributed Systems - FORTE 2005*, volume 3731, pages 443–457, LNCS, 2005. Springer Verlag.
19. CPN group, Univ. Aarhus. cpntools - Computer Tool for Coloured Petri Nets, http://wiki.daimi.au.dk/cpntools, 2008.
20. CS dept. Univ. Hambourg. Renew, http://www.renew.de, 2006.
21. CTR team. Modeling and Analysis Suite for Real-Time Applications, http://mast.unican.es/.
22. C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool Kronos, http://www-verimag.imag.fr/TEMPORISE/kronos/, 2002.
23. M. Ducass and L. Roz. Proof obligations of the b formal method: Local proofs ensure global consistency. In A. Bossi, editor, *Logic Programming Synthesis and Transformation, 9th International Workshop, LOPSTR'99, Selected Papers*, volume 1817 of *LNCS*, pages 10–29. Springer, 2000.
24. A. Duret-Lutz and D. Poitrenaud. SPOT, Spot Produces Our Traces, http://spot.lip6.fr/wiki/.
25. Eiffel software. EiffelStudio - A Complete Integrated Development Environment, http://www.eiffel.com, 2008.
26. Esterel-technologies. SCADE Suite, http://www.esterel-technologies.com/, 2008.
27. S. Evangelista, C. Kaiser, C. Pajault, J-F. Pradat-Peyre, and P. Rousseau. Dynamic tasks verification with quasar. In *Reliable Software Technology - Ada-Europe 2005, 10th Ada-Europe International Conference on Reliable Software Technologies*, volume 3555 of *LNCS*, pages 91–104. Springer, 2005.
28. F. Gilliers, F. Kordon, and J-P. Velu. Generation of distributed programs in their target execution environment. In *Proceedings of the 15th International Workshop on Rapid System Prototyping*, pages 127–134, Geneva, Switzerland, 2004. IEEE Computer Society.
29. J. Gogen and Luqi. Formal methods: Promises and problems. *IEEE Software*, 14(1):75–85, 1997.
30. GreatSPN group. GreatSPN home page, http://www.di.unito.it/ greatspn.
31. G. Holzmann. Logic Verification of ANSI-C Code with SPIN. In *SPIN Model Checking and Software Verification, 7th International SPIN Workshop*, volume 1885 of *LNCS*, pages 131–147. Springer, 2000.
32. G. Holzmann. On-the-fly, LTL Model Checking with SPIN, http://spinroot.com/spin, 2007.
33. G. Holzmann and R. Joshi. Model-driven software verification. In S. Graf and L. Mounier, editors, *Model Checking Software, 11th International SPIN Workshop*, volume 2989 of *LNCS*, pages 76–91. Springer, 2004.
34. G. Holzmann and M. Smith. An Automated Verification Method for Distributed Systems Software Based on Model Extraction. *IEEE Trans. Software Eng.*, 28(4):364–377, 2002.
35. J. Hugues, L. Pautet, B. Zalila, P. Dissaux, and M. Perrotin. Using AADL to build critical real-time systems:Experiments in the IST-ASSERT project. In *4th European Congress ERTS*, Toulouse, Paris, jan 2008.

36. J. Hugues, B. Zalila, L. Pautet, and F. Kordon. From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite. *ACM Transactions in Embedded Computing Systems (TECS)*, oct 2008.
37. A. Kavimandan, A. Narayanan, A. S. Gokhale, and G. Karsai. Evaluating the Correctness and Effectiveness of a Middleware QoS Configuration Process in Distributed Real-Time and Embedded Systems. In *11th International Symposium on Object-oriented Real-time distributed Computing (ISORC'08)*, pages 100–107. IEEE Computer Society, 2008.
38. F. Kordon. Design methodologies for embedded systems: Where is the super-glue? In *11th International Symposium on Object-oriented Real-time distributed Computing (ISORC'08)*, page to be published, Orlando, USA, May 2008.
39. F. Kordon and Luqi. An Introduction to Rapid System Prototyping. *IEEE Transactions on Software Engineering*, 70(3):817–821, 2002.
40. F. Kordon and L. Petrucci. Toward Formal-Methods Oecumenism? *IEEE Distributed Systems Online*, 7(7), July 2006.
41. Labri. FAST - Fast Acceleration of Symbolic Transition systems, http://www.lsv.ens-cachan.fr/fast, 2006.
42. LIP6/MoVe. The CPN-AMI home page, http://www.lip6.fr/cpn-ami/.
43. C. L. Liu and J. W. Layland. Scheduling algorithms for multi-programming in hard-real-time environment. In *Journal of the ACM*, january 1973.
44. M. Spivey. The fuzz type-checker for Z, http://spivey.oriel.ox.ac.uk/mike/fuzz/.
45. P. Madhusudan, editor. *Proceedings of the 9th International Workshop on Verification of Infinite-State Systems (INFINITY'07)*, Electronic Notes in Theoretical Computer Science, Lisboa, Portugal, September 2007. Elsevier Science Publishers.
46. Mathwork. Simulink - Simulation and Model-Based Design, http://www.mathworks.com/products/simulink/, 2008.
47. K. L. McMillan. The SMV System, http://www.cs.cmu.edu/ modelcheck/smv.html.
48. Irwin Meisels and Mark Saaltink. The z/eves reference manual (for version 1.5).
49. OMG. A UML profile for MARTE, Beta 1. Technical Report ptc/07-08-04, OMG, 2007.
50. OPERA Group, Univ. Cambridge. QPME Homepage, http://www.dvs.tu-darmstadt.de/staff/skounev/QPME/, 2007.
51. Praxis Hight Integrity Systems. SPARKAda, http://www.praxis-his.com/sparkada/, 2008.
52. PRISM Team. PRISM - Probabilistic Symbolic Model Checker, http://www.prismmodelchecker.org/, 2008.
53. SAE. *Architecture Analysis & Design Language (AS5506)*. SAE, sep 2004. available at http://www.sae.org.
54. D. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.
55. B. Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003.
56. F. Singhoff. The Cheddar project : a free real time scheduling analyzer, http://beru.univ-brest.fr/ singhoff/cheddar/, 2007.
57. SRI/CSL. PVS Specification and Verification System, http://pvs.csl.sri.com/index.shtml, 2008.
58. Telelogic. Rhapsody, http://www.telelogic.com/products/rhapsody/, 2008.
59. Upenn, Dept of Computer Science. CHARON, http://rtg.cis.upenn.edu/mobies/charon/.
60. UPPAAL Group. UPPAAL, http://www.uppaal.com/.
61. VASY Project - INRIA. Construction and Analysis of Distributed Processes, http://www.inrialpes.fr/vasy/cadp.html, 2005.
62. VASY Project - INRIA. TRAIAN: A Compiler for E-LOTOS/LOTOS NT Specifications, http://www.inrialpes.fr/vasy/pub/traian, 2008.