The STATEMATE Verification Environment
Making It Real

Tom Bienmüller[1], Werner Damm[2], and Hartmut Wittke[2]

[1] Carl von Ossietzky University, 26111 Oldenburg, Germany
Bienmueller@Informatik.Uni-Oldenburg.DE
[2] OFFIS, Escherweg 2, 26121 Oldenburg, Germany
{Damm, Wittke}@OFFIS.DE

Abstract. The STATEMATE Verification Environment supports requirement analysis and specification development of embedded controllers as part of the STATEMATE product offering of I-Logix, Inc. This paper discusses key enhancements of the prototype tool reported in [2,5] in order to enable full scale industrial usage of the tool-set. It thus reports on a successfully completed technology transfer from a prototype tool-set to a commercial offering. The discussed enhancements are substantiated with performance results all taken from real industrial applications of leading companies in automotive and avionics.

1 Introduction

This paper reports on a successful technology transfer, taking a prototype verification system for STATEMATE [2,5] to a version available as commercial product offering. It is only through the significant improvements reported in this paper that key companies are now taking up this technology in an industrial setting. In particular, other approaches such as [15,14], though similar in overall goal, did as of today not reach a comparable level neither in quality of handling nor degree of captured complexity.

The leading players in automotive and avionics base their product developments on well established and highly matured process models, typically variations of the well known V-model originally introduced as a standard for military avionics applications. E.g. [16] reports on the development process and process improvements of Aerospatiale, documenting the benefits of a model based development process. Overall product quality is critically dependent on the familiarity of system and software designers with the established process, and any change, in particular the introduction of a technology completely novel to designers, can potentially cause significant process degradation, thus leading to quality reduction rather than quality enhancements. It is thus essential to tune layout and handling to use-cases well understood and easily appreciated by designers. Particular causes of concern identified in numerous discussions with industrial partners are:

- *scariness of formal methods*: With a typical background in mechanical or electrical engineering, it is prohibitive to expose any of the underlying mathematical machinery to designers. We tried to maintain as much of the "look and feel" of the simulation capabilities of the tool-set, offering in particular pure push button analysis techniques described in Section 2, which completely hide the underlying

verification technology. We spent a significant effort in providing a tight integration with STATEMATE for all verification related activities, as described in Section 4.
- *relation to testing*: A standard topic of discussion is the relation to testing, which typically causes confusion. The techniques are complementary, but model-checking can take over a significant part traditionally handled by testing. In terms of the V-diagram, the so-called virtual V supports model based integration of components of subsystems, and model-checking based verification can replace all model-based testing (if clear interface specifications are given), as well as capture a large share of errors typically detected today in integration testing. We found that test engineers not only accept but appreciate a transition, in which the traditional activity of designing test sequences for properties captured only informally or even mentally is replaced by the process of capturing these in forms of requirement patterns (provided as library), leaving construction of test-sequences to the model-checker.
- *user models*: Test engineers constitute only one out of a number of identified classes of users. While there are differences in company cultures, we feel that a second prominent use-case of model-checking rests in its capabilities as a powerful debugging tool, simply reducing the number of iterations until stable models are achieved, and thus reducing development costs.

Of slightly different nature, though similar in spirit, is the introduction of a methodology for supporting successive refinements of abstractions. We allow to selectively “degrade” accuracy, with which the model-checker tracks computations on selected objects within the cone-of-influence for a given property. In particular, all tests dependent on floats are over approximated, while for objects with finite domain a more refined view of computations can be maintained (see Section 3). This technique is complemented by a symbolic evaluation engine, whose underlying machinery rests on a simplified, but restricted version of the first-order model-checking algorithm documented in [1]—see [1] for a description of the use of these enhancement on automotive applications.

While the prototype system reported in [2,5] was based on the SVE system [10], the current version rests on a tight integration with the VIS model-checker [11] and the CUDD BDD package [17], yielding in average a five time performance boost for model-generation times and a 50% speed-up for model-checking.

Acknowledgments. The development of the STATEMATE verification environment has been a collaborative effort of our research team. We gratefully acknowledge the excellent cooperation with Jürgen Bohn, Udo Brockmeyer, Claus Eßmann, Martin Fränzle, Hans Jürgen Holberg, Hardi Hungar, Bernhard Josko, Jochen Klose, Karsten Lüth, Ingo Schinz and Rainer Schlör, with Peter Jansen (BMW), Joachim Hofmann and Andreas Nägele (DaimlerChrysler – DC), and Geoffrey Clements (British Aerospace – BAe), with the VIS Group, in particular Roderick Bloem, for supporting the integration, Fabio Somenzi for the CUDD BDD package and I-Logix, in particular Moshe Cohen.

2 Model Analysis

The STATEMATE semantics [12] contains modeling techniques which enables simultaneous activation of conflicting transitions or nondeterministic resolution of multiple write accesses to data-items. Even if these mechanisms are captured by the STATEMATE semantics, such properties prove to be errors in later design processes, where the design is mapped to more concrete levels of abstraction, for instance, by using code-generation facilities.

The current version of the verification environment has been extended by *push-button* analysis for STATEMATE designs to be able to verify if some robustness properties are fulfilled by the system under design. These debugging facilities cover

a) simultaneous activation of conflicting transitions,
b) several write accesses to a single data-item in the same step[1], and
c) parallel read- and write-accesses to the same object.

Besides these robustness checks, which are completely automated, the verification environment offers simple reachability mechanisms to drive the simulation to some user provided state or property, for example, a state where some specific atomic proposition holds. Such analysis can be used to verify, for instance, that states indicating fatal errors are not reachable. In the context of testing, reachability related analysis can be used for achieving simulation prefixes.

These use cases require—in contrast to verification activities related to quality acceptance gates for complete Electronic Control Units (ECUs) or subsystems—a white box view of the underlying model, allowing to refer to states and transitions of the model, typically asking for state-invariants.

Implementation and Results. Model analysis is performed using symbolic model-checking as described in Section 1. In order to use model-checking for model analysis, the finite state machine describing the model's behavior is extended automatically by *observers*, which allow to specify robustness properties as atomic propositions p. These propositions are then checked using simple $\mathbf{AG}(p)$ formulae, stating that nothing bad ever happens. If there is a path leading to some state σ with $\sigma \models \neg p$, then that path is a witness for erroneous modeling. This path can be used to drive the STATEMATE simulator as described in [2,4].

Model analysis has been successfully applied to industrial sized applications like a central locking system (BMW; c.f. [7]), a main component of a stores management system of an aircraft (BAe; c.f. [2]), and an application of another leading european car manufacturer. Each application consists of up to 300 state-variables and 80 input-bits. In every model, all types of robustness failures were detected, consuming 300 seconds of analysis time in average on an UltraSparc II, 277MHz, equipped with 1GB of RAM. Note that these results have been achieved on top level designs. If these facilities will—as intended—be used while developing a design, model analysis will be performed much faster as it will be applied mainly to subcomponents of a complete system.

[1] Within STATEMATE, W/W-races are reported disregarding the values to be written simultaneously. Besides such an analysis, the verification environment also provides an enhanced W/W-race detection, also taking into account the values to be assigned, since designers often considered W/W-races wrt. equal values to be not harmful.

3 Propositional Abstraction

Several different approaches have been proposed in the literature to overcome the state explosion problem, which is inherent even to symbolic model checking. Besides techniques for compositional reasoning, the current version of the STATEMATE verification environment provides a simple but powerful abstraction technique, which is offered to the user as special verification method for component proofs. This abstraction method improves the approach described in [2].

State of the art model checkers like VIS [11] compute the cone-of-influence (COI) of a model $\mathcal{M}$ regarding a requirement ϕ before verification is initiated. The COI of $\mathcal{M}$ contains those variables of $\mathcal{M}$, which may influence the truth value of ϕ. As the COI preserves the exact behavior of $\mathcal{M}$ regarding ϕ, it potentially still contains very complex computations for its variables—this complexity often remains too high for successful verification. On the other hand, the COI provides useful information about dependencies of objects within $\mathcal{M}$ when verifying ϕ.

Even if the COI contains all variables which may influence requirement ϕ, some of them can safely be omitted if ϕ must hold for *any* value of these variables. If the exact valuation for some variable x is not required to verify ϕ, also all computations for its concrete representation can be omitted. The propositional abstraction supported in the STATEMATE verification environment provides a mechanism to automatically compute an over-approximation $\mathcal{M}_a$ of $\mathcal{M}$ wrt. a user selected set of variables from within the COI. Every computation required to build $\mathcal{M}_a$ is performed on a higher level language SMI [3], which serves as intermediate format in the STATEMATE verification environment. Note that also the COI can be calculated on that level. Since it is not required to build $\mathcal{M}$ to obtain $\mathcal{M}_a$, the abstraction technique becomes suitable also for models containing infinite objects. Abstracted variables are eliminated and only their influence on other objects is maintained in $\mathcal{M}_a$.

Let x be a variable to be abstracted. Then, each condition b occurring in $\mathcal{M}$ is replaced by $\exists x.b.$[1] Assignments to x are eliminated, and if x is referenced in a right hand side of an assignment $y := t(\dots, x, \dots)$, the assignment is replaced by $y := y\_inp$, where y_inp is a fresh input ranging over the domain of y.[2]. The model checking problem becomes simpler on the abstract model, as any computations regarding x are not further in use.

The verification environment offers an application methodology for propositional abstraction, which is based upon an iterative scheme:

1) compute cone-of-influence of $\mathcal{M}$ regarding ϕ,
2) select set of variables to be abstracted from $\mathcal{M}$ and compute $\mathcal{M}_a$,
   a) if $\mathcal{M}_a \models \phi$, the requirement also holds for $\mathcal{M}$,
   b) if verification fails due to complexity or a (user defined) timeout occurs for $\mathcal{M}_a$, set $\mathcal{M} := \mathcal{M}_a$ and go to 1),
   c) otherwise, analyze error-path to identify needed increase of accuracy.

[1] Since SMI does not provide existential quantification, this is approximated by replacing positive occurrences of propositions referring to x by *true*, while negative ones are replaced by *false*.

[2] [2] reports other possible types of computations for $\mathcal{M}_a$ These types differ in the remaining degree of information about some abstract variable x in $\mathcal{M}_a$.

Results. The abstraction technique and its application methodology have been completely integrated into the verification environment. The abstraction technique itself proves to be powerful for industrial applications. Verification of some non-trivial safety requirements regarding a BAe application (stores management system) with up to 1200 state-bits became possible using this type of abstraction within a couple of seconds. The number of state-bits dropped to 160 in the abstract model. Propositional abstraction has also been successfully applied to a DC application [13], which originally contained four 32-Bit integer and two real variables. The automatically abstracted version used only 80 state-bits and 7800 BDD nodes and model-checking some relevant safety properties was performed successfully within 60 seconds.

4 Integration

The tool-set for Statemate verification is enhanced with a graphical interface which significantly eases its use. All verification related activities can be performed by graphical operations. Compiling the design to the internal representation for the verification [4] as well as model analysis described in Section 2 can be initiated directly from appropriate icons.

The verification environment provides a behavioral view for each activity of the Statemate design. This allows to apply model analysis on each activity separately. Results are recorded in a report. Witness paths found by the analysis operations are translated into simulation control programs [4], which can be used to drive the Statemate simulator.

Verification of specific properties of the design under consideration is supported by a graphical specification formalism integrated in the user interface. For each activity of the design the user can state requirements using predefined specification patterns, which can also be employed to express assumptions about the environment of an activity. Typical properties for verification are offered as a library of patterns, which can easily be instantiated in customized specifications.

The user interface ensures automatic translation of requirements into temporal logic formulae [9], while assumptions about the environment of an activity are compiled into observer automata. Verification is done by adding the observers for the assumptions plus fairness constraints to the model [6] and performing regular model checking using the CTL model checker VIS. This process is managed by the user interface, hiding all control aspects from the user.

Creation of proof obligations and execution of proof tasks is integrated in the user interface. The interface keeps track of verification results. Proof-results are automatically invalidated by changes in the design or modifications of the specification. A fine granular dependency management ensures minimal invalidation. Proofs can be established again wrt. the changes by re-executing the affected proof tasks. Witness sequences are collected and managed by the environment. The user can easily animate runs of the activity violating requirements by using the Statemate simulator. Additionally, a violation can be displayed as set of waveforms.

The user interface offers propositional abstraction for each proof task. The abstraction iterations discussed in Section 3 are recorded and automatically reapplied when the proof task needs to be re-executed, for example, if some changes are made to the model.

Compositional reasoning is guided by the user interface as described in [2]. Specifications of sub-components can be used hierarchically to derive specifications of composed activities of the design. Such structural implications are maintained wrt. the verification results for the sub-activities.

5 Conclusion

We have described key enhancements taking a prototype verification system for STATEMATE to a verification environment now available as commercial offering from I-Logix. As part of the ongoing cooperation, further extensions of the technology are under development, including in particular support for industrially relevant classes of hybrid controller models and support for a recently developed extension of Message Sequence Charts called Live Sequence Charts [8]. Within the SafeAir project, the same verification technology is linked to other modeling tools used by our partners Aerospatiale, DASA, Israeli Aircraft Industries, and SNECMA. In cooperation with BMW and DC we are integrating further modeling tools.

References

[1] Tom Bienmüller, Jürgen Bohn, Henning Brinkmann, Udo Brockmeyer, Werner Damm, Hardi Hungar, and Peter Jansen. Verification of automotive control units. In Ernst-Rüdiger Olderog and Bernd Steffen, editors, *Correct System Design*, number 1710 in LNCS, pages 319–341. Springer Verlag, 1999.

[2] Tom Bienmüller, Udo Brockmeyer, Werner Damm, Gert Döhmen, Claus Eßmann, Hans-Jürgen Holberg, Hardi Hungar, Bernhard Josko, Rainer Schlör, Gunnar Wittich, Hartmut Wittke, Geoffrey Clements, John Rowlands, and Eric Sefton. Formal Verification of an Avionics Application using Abstraction and Symbolic Model Checking. In Felix Redmill and Tom Anderson, editors, *Towards System Safety – Proceedings of the Seventh Safety-critical Systems Symposium, Huntingdon, UK*, pages 150–173. Safety-Critical Systems Club, Springer Verlag, 1999.

[3] Jürgen Bohn, Udo Brockmeyer, Claus Essmann, and Hardi Hungar. SMI – system modelling interface, draft version 0.1. Technical report, Kuratorium OFFIS, e.V., Oldenburg, 1999.

[4] Udo Brockmeyer. *Verifikation von* STATEMATE *Designs*. PhD thesis, Carl von Ossietzky Universität Oldenburg, December 1999.

[5] Udo Brockmeyer and Gunnar Wittich. Tamagotchis Need Not Die – Verification of STATEMATE Designs. In *Tools and Algorithms for the Construction and Analysis of Systems*, number 1384 in LNCS, pages 217–231. Springer Verlag, 1998.

[6] E. Clarke, O. Grumberg, and K. Hamaguchi. Another Look at LTL Model Checking. *Formal Methods in System Design*, 10(1):47–71, February 1997.

[7] W. Damm, U. Brockmeyer, H.-J. Holberg, G. Wittich, and M. Eckrich. Einsatz formaler Methoden zur Erhöhung der Sicherheit eingebetteter Systeme im KFZ, 1997. VDI/VW Gemeinschaftstagung.

[8] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. In *FMOODS'99 IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems*, 1999.

[9] Konrad Feyeraband and Bernhard Josko. A visual formalism for real time requirement specifications. In *Proceedings of the 4th International AMAST Workshop on Real-Time Systems and Concurrentand Distributed Software, ARTS'97, Lecture Notes in Computer Science 1231*, pages 156–168, 1997.

[10] Thomas Filkorn. Applications on Formal Verification in Industrial Automation and Telecommunication, April 1997. *Workshop on Formal Design of Safety Critical Embedded Systems*.

[11] The VIS Group. VIS : A System for Verification and Synthesis. In *8th international Conference on Computer Aided Verification*, number 1102 in LNCS, 1996. VIS 1.3 is available from the VIS home-page: `http://www-cad.eecs.Berkeley.EDU/~vis`.

[12] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. Technical Report CS95-31, The Weizmann Institute of Science, Rehovot, 1995.

[13] Joachim Hoffmann, Hans-Jürgen Holberg, and Rainer Schlör. Industrieller Einsatz formaler Verifikationsmethoden, February 2000. to appear in ITG/GI/GMM-Workshop *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, Frankfurt/Main.

[14] E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical automata as model for statecharts. In *Asian Computing Conference (ASIAN'97)*, number 1345 in LNCS. Springer Verlag, 1997.

[15] E. Mikk, Y. Lakhnech, and M. Siegel. Towards Efficient Modelchecking Statecharts: A Statecharts to Promela Compiler, April 1997.

[16] Francois Pilarski. Cost effectiveness of formal methods in the development of avionics systems at Aerospatiale. In *17th Digital Avionics Systems Conference, Seattle, WA*. Springer Verlag, 1998.

[17] Fabio Somenzi. CU Decision Diagram Package, 1998. CUDD 2.3.0 is available from `http://vlsi.Colorado.EDU/~fabio`.