Verification of Statecharts with SPIN

Erich Mikk, Yassine Lakhnech, Michael Siegel *

1 Introduction

Harel proposed the graphical language statecharts [5, 6] for specifying behavior of reactive systems. Statecharts is supported by the commercial STATEMATE tool [13] for about 12 years now. STATEMATE is a CASE tool with a considerable user community in industry and academia. In STATEMATE, statecharts are used to describe the behavior of the system and they are complemented with other languages for describing the system architecture and functionality (behavioral, structural and functional view [7].) STATEMATE provides a.o. interactive simulation of statecharts specifications and code generation.

Statecharts extend Mealy machines by parallelism and hierarchy. The communication mechanism in statecharts is instantaneous broadcast and the concurrency model is maximal parallelism. The language of statecharts incorporates several abstractions that qualifies it as a specification language for reactive systems. This includes:

- *Uninterpreted events* which are means of communication between the environment and the system under development (or between the parts of the specification).
- Statecharts is related to the *synchrony hypothesis* ([2]) which states that the controlling hardware/software is infinitely faster than the environment it controls.

Such abstractions make statecharts an attractive candidate for exhaustive automatic analysis with traditional model checking techniques that cope with finite state systems. A systematic way of exploiting existing model checkers for the automatic verification of statecharts is a translation of statecharts to the input language of a model checker.

SPIN [9, 19] is a verification package that includes one of the most efficient linear temporal logic model checkers. SPIN is an explicit state model checker with a nested depth-first search algorithm [11] with various state compression

*Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik, Preußerstr. 1-9, D24105 Kiel, Germany, e-mail: {erm,yl,mis}@informatik.uni-kiel.de

techniques [8, 12]. SPIN has an implementation of partial order reduction techniques [10]. The input language of SPIN is PROMELA which offers a wide variety of constructs for the description of distributed systems.

The challenge of translating statecharts to the input language of any model checker are the following two aspects:

Correctness of the translation. To justify the correctness of the translation, we underpin it with formal semantics of statecharts. We proposed an intermediate format of *extended hierarchical automata* (EHA) [16] as a framework for describing the semantics of statecharts. EHA are suitable to cope with interlevel transitions in a structurally clean way. The translation to PROMELA/SPIN is two-phase: first, statecharts are translated to EHA, and second, extended hierarchical automata are translated to PROMELA.

Performance. Given the target model checker, the translation scheme needs to be adjusted in order to achieve better verification performance. In [17] we discuss optimizations performed by our translator to PROMELA/SPIN.

In the current paper we focus on practical use of our translator together with SPIN and we discuss following issues:

Feasibility of LTL model checking with SPIN. Can statecharts with non-trivial size be analyzed with SPIN? What are the run-time figures?

Feasibility of counterexamples. Can counterexamples as generated by SPIN be interpreted in terms of the original specification?

Limiting factors. What is the limiting factor of model checking PROMELA programs generated by our translator? Can statecharts specification design help to alleviate the state explosion problem?

On the use of analysis techniques. Apart from LTL model checking, SPIN offers analysis techniques (e.g., search for deadlocks, starvation). Can those techniques also be exploited for statecharts?

Related work in the area of model checking statecharts is too extensive to be discussed here. Probably the most closely related work is that of Brockmeyer and Wittich [3] because they investigate the same version of statecharts. They report on translating almost the whole language of statecharts as supported by STATEMATE to the input language of a symbolic model checker [4].

2 Feasibility of model checking

In order to demonstrate the feasibility of the translator and the SPIN verification system for model checking statecharts, we use a well-know case study.

Case study

The production cell case study [15] was launched by Forschungszentrum Informatik in Karlsruhe, Germany, to demonstrate benefits of formal methods for industrial applications. Up to now, 36 contributions including 2 with statecharts are reported [18]. We used the specification prepared by the the group of Prof. W. Damm (University of Oldenburg) [14, 15]. For brevity we restrict the discussion to the verification figures only (very detailed explanation of this specification can be found in [14, 15]).

The statecharts specification contains 30 parallel automata, 30 events are used to synchronize components of the model. The number of variables needed to encode states and events in PROMELA is 137 (60 boolean and 77 enumeration type variables). The reachable state space is $3.08191e+06$ states as counted by SPIN version 3.2.3.

Formalization of properties for translator-generated models

For PROMELA models the LTL properties are formulated in terms of variables of the model. In our case the PROMELA model is generated automatically. Obviously we want to formulate the properties in terms of the original statecharts without knowing much detail about the translation schema and the resulting model. Here we discuss how this is achieved.

State transitions of statecharts (also called "step") from one status to the next may include taking several transitions in the statecharts. Due to the structural nature of our translation, we translate statecharts transitions into PROMELA code. In order to implement the semantics correctly, the PROMELA code must display its state only at step boundaries, thus the successive computation of one step must be hidden from the property to be verified. This is done with enclosing the step computation within one `atomic` statement. Since the never-claim may reference any variables of the model under consideration, the property to be verified may be formulated in terms of PROMELA variables. Here the situation is quite favorable: we have Promela variables that implement the control states and the events of the original statechart.

Our translator generates propositions that simplify the formalization of properties. According to the semantics of statecharts, a computation is a sequence of statuses, and a status exposes which control states of the statechart are active and which events are available. Propositions are interpreted with respect to the current status. Our translator generates for a control state ST a proposition `in_ST`. The proposition is true iff the state ST is active in the current status. Similarly, for an event EV a proposition `gen_EV` is generated that is true iff the event is available in the current status. With these propositions available, the user needs only to know the above naming convention in order to formalize properties that can be checked with SPIN.

Verification of safety properties

As an example, we give the verification figures of one safety property here. One group of safety properties in this example considers the interplay between various devices: those devices should not collide. Our experiment verifies that no collision is possible in the statechart model. The experiment is done on an Sparc Ultra 1 of 1GB memory. Here are the statistics of the state space analysis.

- Default search: 266MB , 1:37:21h,
- Graph encoding technique [12]: 111MB , 2:34:08h

These figures show that there is a trade-off between the default search and graph encoding technique: whereas the default search is fast and consumes more memory, the graph encoding technique is slower but reduces the memory requirements considerably.

Further experiments showed that we could not benefit from the following techniques:

Partial order reduction. In [17] we motivated that generation of sequential code (there is a single PROMELA process in the models generated by our translator) prevail over parallel code (multiple processes). This is so even if the possibility of partial order reduction as implemented in SPIN is also considered. As a result, our models can not benefit from this reduction technique. Intuitevly, the current heuristics of partial order reduction as implemented in SPIN can cope with asynchronous systems only. But statecharts is a representative of the synchronous model of computation.

Recursive indexing method An experiment with the recursive indexing method ([8]) turned out to be worse in time and space than the default search. The reason is that all data is global in our models, but recursive indexing exploits the fact that data is associated with processes (like local variables and communication channels).

Verification of liveness properties

As an example we check if the production cell specification is free from starvation cycles. The designer of the specification included 15 self-loops into the specification that model time (in the sense that taking a self-loop prevents progress). In order to check for meaningful starvation cycles (those executing a self loop are trivial ones), we have to tackle progress. We selected 15 control states of the statechart to be a "progress state": the protocol must pass at least one of them infinitely often in order to make progress. And there is a starvation loop if there is a loop passing non of them.

We formulated a verification task

```
[] <> good_state
```

where `good_state` is a disjunction of the above described 15 progress states: `good_state` holds iff the system is at least in one of its progress states. A SPIN run that checks this property results in the counterexample after 5 minutes. The trace prefix that leads to the starvation loop consists of 13398 statuses (states of the statecharts semantics). The starvation loop itself consists of 2 statuses.

Now we focus on the interpretation of this counterexample in terms of the original specification. Therefore we consider the start status of the starvation loop. This status told us that the resources of the specification were exhausted (the system was saturated) and the starvation loop is indeed a non-progress loop. In order to remedy this situation, one has to change the specification. For example, one can limit the number of blanks in the system (blanks consume the resources).

Comparison with other contributors of the case study

We tried to verify safety properties with the dynamic test tools of STATEMATE (a reachability analysis tool, which is no longer part of the distribution). The case study was infeasible for this tool: we didn't manage to get any results within a runtime of less than one day.

The verification figures are incomparable with those of other contributions ([15]) because of different modeling approaches. The closest modeling is from P. Kelb [14], but he uses a different semantics of statecharts. A comparison of his figures with ours would not be very meaningful. For example, he didn't verify the above described liveness property. We don't know whether his specification also suffers from the problem described above (a non-trivial starvation loop).

Feasibility of counterexamples

Besides the ability of verification (proving the absence of errors) a very important property of model checkers is to give meaningful information why the property under consideration does not hold. Since the original statecharts specification was already verified, we didn't not expect to find errors there.

Instead, we conducted following experiment. As usual in LTL, in order to prove reachability of a proposition `P`, one checks whether the negation of `P` holds in every state:

```
[] ! P
```

If this check fails, the counterexample witnesses that `P` is reachable. In the production cell case study, we have 30 communication events. It is desirable, that every event is reachable. So we expect to get 30 counterexamples, one for every event, that witness the reachability of this event.

Again, we want to interpret SPIN-generated counterexamples in terms of the original specification. In order to do so, the translator adds `printf` statements into the generated model. With the help of those, we can obtain an

ASCII-representation of the statechart computation (a sequence of statuses, as described above). The user may examine this trace or alternately, use our further tools that translate this information into a guided simulation within the STATEMATE tool. This feature makes it quite comfortable to examine the counterexample within STATEMATE (one can execute the counterexample step-by-step, or up to certain breakpoints, or trace back in the counterexample). Hence one can replay counterexamples in terms of the original statechart.

Event	Default	Shortest	Event	Default	Shortest
A1LOC_AT_PRESS	84	54	A1LOC_AT_SAFE	37	5
A1LOC_AT_TAB	44	14	A1_AT_PRESS	76	46
A1_AT_TAB	40	10	A1_CATCH	46	29
A1_DROP	86	56	A2LOC_AT_DB	144	114
A2LOC_AT_PRESS	147	117	A2LOC_AT_SAFE	5	5
A2_AT_DB	72	42	A2_AT_PRESS	69	39
A2_CATCH	149	119	A2_DROP	166	134
AT_DB	12	10	AT_FB	186	162
AT_OK4DB	20	18	AT_OK4FB	16	14
AT_TOP	9	5	CRANE_CATCH	171	153
CRANE_DROP	243	175	DB_IN_LB	169	137
FB_IN_LB	7	7	PRESS_BOTTOM	103	73
PRESS_MIDDLE	5	5	PRESS_TOP	96	66
TABLE_DOWN	22	5	TABLE_OK4FB	24	5
TABLE_OK4ROB	35	17	TABLE_TOP	33	17

Figure 1: Event witnesses.

A further issue of interest is the length of counterexamples. In Table 1 we summarize the lengths of traces that witness that a certain event occurs. Those events give feedback to the specifier, that the statecharts model indeed does something useful. In order to judge about the quality of these witness traces, we took the effort to generate the shortest counterexamples as well. In the default case, the overall generation time for these event witnesses (for each event, generation of the property, generation of `pan`, the run of `pan` and simulation run) is less than 10 minutes. The generation of shortest counterexamples lasted 2.5h that can be considered feasible as well. Memory is not a limiting factor in this case because the traces are very short.

In Table 1, the default counterexamples are quite good approximations of the shortest ones.

We also considered to find a shorter prefix to the counterexample we found during the verification of the liveness property as discussed above. The default prefix was 13398 statuses long. A SPIN run of 6 hours didn't show any results. (We didn't try approximation option yet.) Simulation of counterexamples of this length in STATEMATE is infeasible with our current implementation. A dedicated implementation in the STATEMATE tool would remedy this shortcoming.

However, the shortest counterexamples generated by SPIN are not necessarily the shortest ones with respect to the semantics of statecharts. The heuristic for

finding the shortest counterexamples in SPIN minimizes the resources for finding counterexamples. These resources (depth of depth-first search) are needed for storing states visible to the property and intermediate states (e.g, those that come from inside `atomic` statements). But it is not the case that the shortest trace in terms of visible states and intermediate states yields the shortest trace when projected to visible states only.

Non-determinism as limiting factor

The limiting factor of model checking is the state explosion problem which is related to the number of parallel components in the model and ranges of variables. Here we discuss an observation that the state explosion of statecharts as generated by our translator has a specific nature. Therefore consider the verification figures of the safety property mentioned above (different devices of the production cell do not collide):

```
State-vector 44 byte, depth reached 3435806, errors: 0
Minimized Automaton:    121465 nodes and 496593 edges
3.08191e+06 states, stored
1.47914e+07 states, matched
1.78733e+07 transitions (= stored+matched)
8.19073e+08 atomic steps
hash conflicts: 0 (resolved)
(max size 2^19 states)

Stats on memory usage (in Megabytes):
147.932 equivalent memory usage for states (stored*(State-vector + overhead))
9.591   actual memory usage for states (compression: 6.48%)
70.000  memory used for DFS stack (-m3500000)
111.984 total actual memory usage
```

We used the minimized automaton option and the length of the depth-first stack (DF stack) is close to the minimal possible. We observe that the DF stack consumes more than 7 times more memory than the actual state space stored in the minimized automaton. Consequently, one should try to minimize the DF stack because it is more memory-intensive with this storage technique. One possibility to reduce the depth of DF stack is to reduce the number of stack frames that are caused by intermediate states within an `atomic` statement. This should be our main concern because the whole code is marked with one atomic label.

As we discussed in [17] the number of stack frames can be reduced if sequences of PROMELA statements can be marked as being deterministic (`d_step`). The translator implements few simple heuristics for detecting deterministic automata and it adjusts the code with the help of this information. This leads to considerable reduction of the length of the DF stack for this example.

In further work, one could extend this heuristics with an extension in statecharts language for marking deterministic automata (similarly to `d_step` in

PROMELA). Then the translator could group together deterministic automata into one `d_step` statement.

3 Other possibilities of analysis

Check for absence of deadlocks

SPIN allows to search for deadlock states in the PROMELA model. In the deadlock state no executable statements exists, but the resources (e.g, memory) are still available. This feature is of no practical use for models generated by our translator because due to the semantics of statecharts and our translation, such states do not exists in the resulting PROMELA models. The semantics of statecharts tells that if non of the transitions of statecharts can be executed in some state, the statechart stutters in this state. This stuttering is explicitly modeled in the PROMELA programs generated by our translator. Therefore, a check for absence of starvation is more interesting.

Check for absence of starvation using progress labels

In order to check for absence of starvation, one has to determine progress states in the PROMELA model. This can be performed manually on the models generated by our translator. However, progress labels added to the code generated by the translator may have no sensible meaning in the context of statecharts. Because progress labels mark *control states* of the PROMELA model, but we are interested only in the values of variables as they encode statuses of statecharts. **We suggest that progress labels are not used in the code generated by our translator.**

Absence of starvation can be formulated as LTL property (an example is discussed in the subsection about verification of liveness properties), and hence, the verification can manage without them.

Approximation techniques

If the exhaustive analysis fails due to resource limitations, one may use approximation techniques, for example bitstate analysis. Up to now we have conducted no experiments with our models and approximation techniques.

4 Brief overview of the translator and the accompanying tools

This paper does not cover our tools which make the use of our translator from statecharts to PROMELA comfortable, but we made them available to the public ([1]).

Model extractor. Besides the translator, we have a model extractor that allows to use STATEMATE as graphical user interface for drawing statecharts. Our model extractor parses statecharts for later use in the translator. It also resolves name conflicts and merges off-page statecharts into the parent chart (this allows to analyze statecharts that are spread over several editors in terms of STATEMATE).

Verification back-end. We also implemented a SPIN back-end that allows to translate counterexamples back into terms of the original statecharts specification. As discussed above, this facility allows to automatically replay SPIN-generated counterexamples in STATEMATE. With this back-end, the translation described in this paper is hidden from the user such that the user does not need to learn PROMELA in order to use SPIN for model checking statecharts. In particular, it hides intermediate results due to the successive computation of one step.

Abstract syntax. For those who use other graphical editors for drawing statecharts, we have documented our *abstract syntax,* such that the translator can be interfaced with other editors for drawing statecharts. In this case either ASCII or C-code level interface is available.

The user's manual available from [1] contains information on how to use these tools.

5 Conclusion and further work

We discussed model checking statecharts with SPIN using automatic translation from STATEMATE statecharts to the input language of SPIN. Our experiment with a well-know case study showed that the verification is feasible. Performance figures are acceptable. We motivated that with little effort we managed to make verification comfortable: properties may be formulated in terms of the original specification and the verification results (especially counterexamples) may be interpreted in terms of the original specification. We explained that not all parts of the SPIN verification system benefit the model checking of statecharts. Further, we explained how non-determinism limits the practical model checking with our approach.

Further work includes extension of the sub-language of statecharts by timing primitives and data manipulation. We also plan to investigate compositionality and abstraction which help to scale automatic analysis techniques to bigger examples.

We suggest that the users of SPIN consider our tool-set [1] for experiments of model checking statecharts.

Acknowledgments

We thank Gerard Holzmann for many fruitful discussions on the topics discussed in this paper. Many of the optimizations of PROMELA code described here are suggested by him. We appreciate the help of Peter Kelb who made the specification of production cell available and Bernd Weber for adapting it to STATEMATE. He also wrote the model extractor and the verification back-end. The translator was written using the VDM Domain Compiler and we thank Uwe Schmidt and Hans-Martin Hörcher for providing this tool. Erich Mikk thanks Bell Labs, where he worked on this topic in the summer of 1997.

References

[1] MOCES: MOdel ChEcking Statecharts. `http://www.informatik.uni-kiel.de/~erm/MOCES/`

[2] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[3] U. Brockmeyer and G. Wittich. Tamagotchis Need Not Die – Verification of STATEMATE Designs. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'98)*, number 1384 in LNCS, pages 217–231. Springer Verlag, 1998.

[4] T. Filkorn and SIEMENS AG. Applications of Formal Verification in Industrial Automation and Telecommunication. In *Proceedings of Workshop on Formal Design of Safety Critical Embedded Systems*, 1997.

[5] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[6] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, Oct 1996.

[7] D. Harel and M. Politi. *Modelling Reactive Systems with Statecharts: the Statemate Approach.* i-Logix, Inc, 1996.

[8] G.J. Holzmann. State Compression in Spin. *Proc. Third Spin Workshop*, April 1997.

[9] G.J. Holzmann. The Model Checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[10] G.J. Holzmann and D. Peled. An improvement in formal verification. In *Proc. FORTE94*, Berne, Switzerland, October 1994.

[11] G.J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. *Proc. Second SPIN Workshop*, 1996.

[12] G.J. Holzmann and A. Puri. A Minimized Automaton Representation of Reachable States. Manuscript, 1997.

[13] iLogix Web Site. `http://www.ilogix.com/company/company.htm`

[14] P. Kelb. *Abstraktionstechniken für Automatische Verifikationsmethoden.* PhD thesis, Universität Oldenburg, 1995.

[15] C. Lewerentz and T. Lindner. *Formal Development of Reactive Systems: Case Study Production Cell.* Number 891 in LNCS. Springer Verlag, 1995.

[16] E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical automata as model for statecharts. In *Asian Computing Science Conference (ASIAN'97)*, volume 1345 of *LNCS*. Springer Verlag, December 97.

[17] E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann. Implementing Statecharts in Promela/SPIN. In *Proceedings of "Workshop on Industrial-strength Formal specification Techniques" (WIFT'98)*, To appear. IEEE, 1998.

[18] Production Cell Web Site.
`http://www.fzi.de/prost/projects/production_cell/ProductionCell.html`

[19] SPIN Web Site. `http://netlib.bell-labs.com/netlib/spin/whatispin.html`