Using Promela in a Fully Verified Executable LTL Model Checker*

René Neumann

Technische Universität München, rene.neumann@in.tum.de

Abstract. In [4] we presented an LTL model checker whose code has been completely verified using the Isabelle theorem prover. The intended use of the checker is to provide a trusted reference implementation against which more advanced checkers can be tested. However, in [4] the checker still used an ad-hoc, primitive input language.
In this paper we report on CAVA, a new version of the checker accepting inputs written in Promela. We report on our formalization of the Promela semantics within Isabelle, which is used both to define the semantics and to automatically generate code for the computation of the state space. We report on experiments on standard Promela benchmarks comparing our tool to SPIN.

1 Introduction

Nearly every hand-written software in the ecosystem suffers from bugs. This is almost inevitable when the software is geared towards high performance and therefore uses highly complicated algorithms and data structures which are not easily provable.

On the other hand, it is in general not feasible to thoroughly prove correctness of the code itself, though exceptions exist [10]. But other measures, like model checkers, allow to nevertheless increase trust in the software. As such, the tool is in the role of a trust-multiplier. Hence, their verdict must not be wrong. Now the recursion begins – or as [17] puts it: "*Quis custodiet ipsos custodes?*" – "Who will watch the watchmen?"

Different approaches to tackle this problem exist (an overview is given in [6]). We proposed a pragmatic solution in a previous paper [4]: a verified *reference implementation* of an LTL model checker for finite-state systems à la SPIN [9], *CAVA*. This model checker follows the well-known automata-theoretic approach [19]: Given a finite-state program P and a formula ϕ, two Büchi automata are constructed that recognize the executions of P, and all potential executions of P that violate ϕ, respectively. Then the product of the two automata is computed and tested on-the-fly for emptiness.

To prove full functional correctness of executable code, we define the program in the logic HOL (roughly: combination of functional programming language with logic) of the interactive theorem prover Isabelle [12]. After proving the

* Research supported by DFG grant CAVA, *Computer Aided Verification of Automata*

program correct in Isabelle/HOL, ML (or OCaml, Haskell or Scala) code can be generated *automatically* from those definitions [8]. Using refinement, abstract but proof-friendly definitions can be rewritten as efficient, more complex variants, while still preserving correctness (cf. [4,11]).

In our implementation *CAVA*, the four parts (system construction, LTL-to-Büchi conversion, product construction, emptiness check) are not fixed. Instead, an interface in form of proof obligations and types is defined. Anything fulfilling those obligations and exposing functions of the specified type can therefore be used for the corresponding part. LTL-to-Büchi conversion and product construction currently offer no alternative, using the algorithm of Gerth et al. [7] and the standard on-the-fly construction, respectively. Emptiness checking is currently implemented by different flavors of nested depth-first search algorithms [3,16]. For system construction, we offer the modeling languages *Boolean programs* [4] and Promela [2]. The first is an ad-hoc approach and fairly limited in its expressiveness, while Promela is a powerful, widely used language.

This paper will focus on the novel support for Promela models. By sharing the modeling language with SPIN, we strengthen our position as a reference implementation. Further, we enhance the comparability of timing results by removing the problem of different state space sizes, which complicates comparisons [4]. What is more, our work also serves as a formalization of Promela semantics. This allows implementation of optimizations proven to preserve semantics, but also serves as a source of documentation.

We will give a short overview about our Promela support in Section 2 with more formalization details following in Section 3. In Section 4 we will outline the trustworthiness of the resulting program. We will then conduct experiments and elaborate on the results in Section 5. In the final Section 6 we talk about (possible) future work.

The tool and all supporting material, including the ML code, can be found online at `http://cava.in.tum.de/VSTTE14`.

2 Promela

Promela [2] is a modeling language, mainly used in the model checker SPIN [9]. It offers a C-like syntax and allows to define processes to be run concurrently. Those processes can communicate via shared global variables or by message-passing via channels. Inside a process, constructs exist for non-deterministic choice, starting other processes and enforcing atomicity. It furthermore allows different means for specifying properties: LTL formulae, assertions in the code, never claims (i. e. an automata that explicitly specifies unwanted behavior) and others.

Some constructs found in Promela models, like `#include` and `#define`, are not part of the language Promela itself, but belong to the language of the C preprocessor. SPIN does not process those, but calls the C compiler internally to process them. In CAVA we do the same.

Though there are approaches for giving a formal semantics of Promela [20,5,18], none of them shows that its definition matches reality. Moreover, some refer to outdated versions of the language.

Therefore, observing the output of SPIN and examining the generated graphs often is the only way of determining the semantics of a certain construct. This is complicated further by SPIN unconditionally applying optimizations. For the current formalization we chose to copy the semantics of SPIN, including the aforementioned optimizations. For some constructs, we had to restrict the semantics, i. e. some models are accepted by SPIN, but not by CAVA. Those deviations are:

- `run` is a statement instead of an expression. SPIN here has a complicated set of restrictions unto where `run` can occur inside an expression. The sole use of it is to be able to get the ID of a spawned process. We omitted this feature from CAVA to guarantee expressions to be free of side-effects.
- Variable declarations which got jumped over are seen as not existing. In SPIN, such constructs show surprising behavior:
  `int i; goto L; i = 5; L: printf("%d", i)` yields 0, while
  `goto L; int i = 5; L: printf("%d", i)` yields 5.
  The latter is forbidden in CAVA (it will get rejected with "unknown variable i"), while the first behaves as in SPIN.
- Violating an `assert` does not abort, but instead sets the variable `__assert__` to true. This needs to be checked explicitly in the LTL formula. We plan on adding this check in an automatic manner.
- Types are bounded. Except for well-defined types like booleans, overflow is not allowed and will result in an error. The same holds for assigning a value that is outside the bounds. SPIN does not specify any explicit semantics here, but solely refers to the underlying C-compiler and its semantics. This might result in two models behaving differently on different systems when run with SPIN, while CAVA, due to the explicit bounds in the semantics, is not affected.

Additionally, some constructs are currently not supported, and the compilation will abort if they are encountered: `d_step`[1], `typedef`, remote references, bit-operations, `unsigned`, and property specifications except `ltl` and `assert`. Other constructs are accepted but ignored, because they do not change the behavior of a model: advanced variable scoping, `xr`, `xs`, `print*`, priorities, and visibility of variables.

Nonetheless, for models not using those unsupported constructs, we generate the very same number of states as SPIN does. An exception applies for large `goto` chains and when simultaneous termination of multiple processes is involved, as SPIN's semantics is too vague here.

[1] This can be safely replaced by `atomic`, though larger models will be produced then.

```
record pState =
  pid      -- "Process identifier"
  vars     -- "Local variables"
  pc       -- "Program counter"
  channels -- "Reference local channels"
  s_idx    -- "Reference program"
```

```
record gState =
  vars      -- "Global variables"
  channels  -- "Channels are always global"
  timeout   -- "No process can make transition"
  procs     -- "List of all running processes"
```

```
record edge =
  cond   -- "Necessary condition"
  effect -- "Effect on states"
  target -- "Next state"
  prio   -- "Priority"
  atomic -- "Atomicity information"
```

Fig. 1: Structure definitions

3 Formalization and Implementation

Any formalization of a program needs to specify three things: How to encode the program structure (i. e. the operations and the control flow), how to encode the program state, and how to compute the set of successor program states (i. e. execute a program).

A Promela program does not consist of a single thread of action, but instead consists of multiple processes which run independent of one another except for when they interact. This is reflected in our formalization.

The program structure of Promela is represented by a set of transition systems: For each process p_i, we define $T_i = (S_i, I_i \in S_i, \delta_i \subseteq S_i \times E)$, where $S_i \subseteq \mathbb{N}$ (the set of program points) and E is the set of all records of type `edge`, as defined in Fig. 1, i. e. δ_i is the transition relation (the target is encoded in `edge`). I_i then is the initial program point for this process.

The program state is encoded in two different types of environments, also given in Fig. 1: `gState` for the global state and `pState` for the state of each process. Naturally, the global state contains the set of all current `pState`s (field `procs`).

The program is constructed from an abstract syntax tree (AST), enriched with semantic information (e. g. variables annotated by their type), which gets translated into the aforementioned set of transition systems, and an initial `gState` structure.

Calculating the next steps of execution is formalized by the *successor* function (SPIN calls it *semantic engine*), as required by CAVA to serve as a system implementation. For a given configuration and program, it specifies the set of all possible transitions and resulting states. For each process, the set of all edges from the current state is taken into account. The `effect` of each edge whose `cond` evaluates to true under the current environment is then applied to yield a new environment. In case of an `atomic` block, successors are computed until either no further transition is possible (atomicity is lost), or the block is left. Only the last environment is then presented in the result set. As this part is based on SPIN, more information can be found in [9, Chap. 7].

```
stmntToState (StmntAssign v e) (lbls, pri, pos, nxt, _) =
  ([[⦇cond = ECTrue, effect = EEAssign v e, target = nxt, prio = pri, atomic = NonAtomic⦈]],
  Index pos, lbls)

stmntToState (StmntCond e) (lbls, pri, pos, nxt, _) =
  ([[⦇cond = ECExpr e, effect = EEId, target = nxt, prio = pri, atomic = NonAtomic⦈]],
  Index pos, lbls)"

stepToState (StepStmnt s (Some u)) (lbls, pri, pos, nxt, onxt, _) = (
let
 (* the 'unless' part *)
 (ues,_,lbls') = stmntToState u (lbls, pri, pos, nxt, onxt, True);
 (* 'u' is the guard for the whole unless; 'ues' the rest *)
 u = last ues; ues = butlast ues;
 pos' = pos + length ues;
 (* find minimal current priority *)
 pri' = min_prio u pri;
 (* the main part –
    priority is decreased, because there is now a new unless part with higher prio *)
 (ses,spos,lbls'') = stmntToState s (lbls', pri' − 1, pos', nxt, onxt, False);
 (* add an edge to the unless part for each generated state *)
 ses = map (List.append u) ses
in (ues@ses,spos,lbls''))
```

Fig. 2: Construction of transition system (excerpt)

As noted, the translation into a set of transition systems requires an enriched AST. This is achieved in two steps: A hand-written SML parser translates the Promela source into an abstract syntax tree. This data structure is then enriched in Isabelle with the semantic information and some constructs (e. g. `for`-loops) get replaced by semantically equal parts (de-sugaring). This step allows to keep the *semantic engine* more concise and explicit, also straightening proofs.

In Fig. 2 we show the construction of the edges for three exemplary nodes in the enriched AST: The first two, `StmntAssign` and `StmntCond`, are representative examples for most of the AST-nodes: A specific condition and effect are set, and control passes to the next statement. It is to note, that `cond` for the first node and `effect` for the second one each resemble a no-op, as expected. The third example is an `{s} unless {u}` construct, which is one of the more complicated control structures in Promela: from each step in the sequence `s`, control can go the the `unless`-part `u` as soon as the first expression in `u` becomes true. In general, all constructs influencing the control flow (e. g. `do` or `if`) are complex. To a great degree, this is due to SPIN's semantic trying to minimize the use of intermediate states, something commonly happening with nested loops – even more when (nested) `unless` is involved. Another complication for constructing the control flow originates from atomicity, which can be passed between processes (by handshakes), lost (on blocks), or chained (by `goto`).

In Fig. 3 we display snippets from the evaluation function for the condition on the edges. Again, this is in most cases rather straightforward. Those examples amount to: expressions must evaluate to something non-zero; spawning a new process requires the number of currently running processes to be below some

```
evalCond (ECExpr e) g l ↔ exprArith g l e ≠ 0
evalCond (ECRun _) g l ↔ length (procs g) < 255
evalCond (ECSend v) g l ↔ withChannel v (λ_ c. case c of
                                          Channel cap _ q ⇒ length q < cap
                                        | HSChannel _ ⇒ True) g l
```

Fig. 3: Evaluation of conditions (excerpt)

```
evalEffect (EEAssign v e) _ g l = setVar v (exprArith g l e) g l
evalEffect (EERun name args) prog g l = let (g,_) = runProc name args prog g l in (g,l)
evalEffect (EEAssert e) _ g l =  if exprArith g l e = 0
                                 then setVar assert 1 g l
                                 else (g,l)
```

Fig. 4: Evaluation of effects (excerpt)

upper bound[2]; and for sending something over a channel, the capacity of this channel must not be exhausted.

The structure for evaluating the effects is similar, as shown in Fig. 4: a variable is set to the correct value; a new process is started; the `__assert__` variable is set, if the expression is true. For sending and receiving (not shown) more effort is necessary, stemming mostly from the different variants and from the fact, that receiving can compare values, evaluate variables, and set variables at the same time.

The formalization presented is, as explained, currently quite SPIN-centric and therefore in parts too specific and concrete, especially concerning optimizations. But the current work now allows to abstract into a more concise formalization. Thereafter, the current implementation can be shown to adhere to the same semantics. As was done in the other parts of CAVA, further optimizations can then be applied. Using the refinement approach [11], this does not affect the abstract formalization.

To give some insight about the size of development: The Isabelle theories regarding Promela span about 2100 lines, with the generated code being about 3100 lines of ML. Additional 2400 lines are added by the parser.

4 Trustworthiness

When speaking about something being *verified*, one needs to state explicitly which properties the result is guaranteed to have. Also, except when noted otherwise, some parts are assumed to be correct, like compiler, hardware, and operating system.

For CAVA, the property to hold is the correctness theorem about the model checker: A lasso is found iff the property does not hold for the system (cf. [4]), and, if existing, the lasso is a counter-example. As some parts of CAVA are programmed directly in SML (especially the user-interface and the parsers), no correctness assumption can be made about them. For example, CAVA may return

[2] A necessary condition for a finite state-space.

a correct lasso, but the output might still be erroneous, due to a mistake in the printing function. But as those parts are a) easily checkable by hand and b) no useful correctness properties can be shown for them (how would one express, that the result of a parser is correct without adding another unproven layer on top?), we still claim the result to be verified.

It remains to show, that the properties hold on the generated code. By design of Isabelle/HOL as an LCF-style theorem prover, every proof-construction is done by primitives of a small trusted kernel [12]. Therefore the proofs in it are correct deductions in the logic and the properties hold. The task of the code-generator [8] then is to translate the definitions from HOL syntax to the target syntax; one can see it as a pretty-printer. Because of this translation on a syntactic level, the properties also hold on the exported code.

What is not covered here is the actual formalization of the properties itself, i. e. whether the HOL term represents the informal claim one is expecting. As [15] lays down in detail, this problem is inherently unsolvable in an automatic fashion and can only be checked by the interested human itself.

5 Evaluation

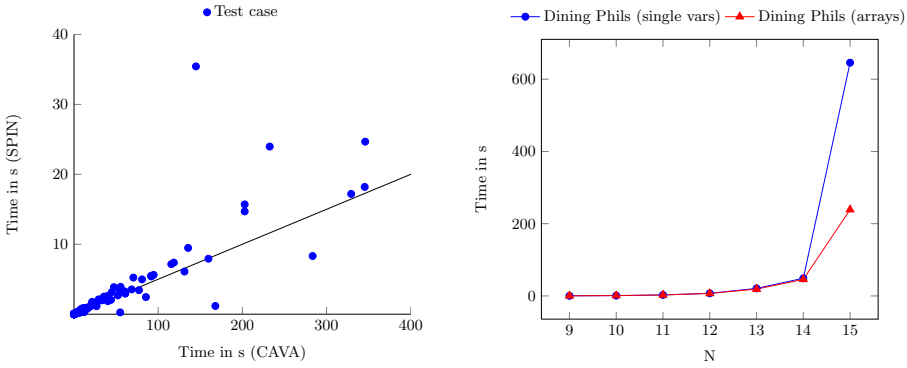
With the support for Promela, it is now possible to test the very same models in both SPIN and CAVA. For this, we used models from [13,1], with some minor modifications to match modern Promela syntax. The tests were performed on a Core i7 with 2.7 GHz, memory being hard-limited to 6.5 GB. Also, a timeout of 800 seconds was set for each run.

CAVA was compiled with MLton 20130715. SPIN (version 6.2.5) was used without optimizations, especially partial-order reduction: `spin` was run with `-o1 -o2 -o3` and the code compiled with `-DNOREDUCE`. During the benchmark, SPIN's search depth was set to $6 * 10^7$ (`-m60000000`).

Further, `-Dd_step=atomic` was passed to both SPIN and CAVA, replacing `d_step` blocks by `atomic` blocks, as the former is not supported by CAVA. Since `d_step` is an optimized and restricted form of the latter (collapsing the sequence into one state), this is semantically sound, but influences the size of the state space.

The benchmark consists of 306 single tests, 4 of which got removed, as they contained failing asserts which CAVA ignores by default (cf. Sect. 2). Further, 50 tests included features not supported in CAVA, 77 led to failures in SPIN (most often out-of-memory and exhausted search depth), 94 timed out on CAVA (a test may occur in multiple of those categories). In total 157 tests performed successfully on both tools. To ensure a complete search of the state space the property used together with those tests is `G true`. Each test was run 5 times, the worst and best time removed and the remaining three averaged. Two timed out runs mark the whole test as timed out.

This benchmark shows, that overall CAVA is about 20 times slower than SPIN. Fig. 5a plots the results of the benchmark: the line represents $t_{\text{SPIN}} = 20t_{\text{CAVA}}$, so anything above represents a test where CAVA was less than 20 times slower



(a) G true benchmark on 157 tests

(b) Comparison array vs. single variables

Fig. 5: Benchmark results

than SPIN (dots below analogously). Tests on which it timed out, had a mean run time of 89.18 seconds in SPIN, lying far above $\frac{\text{timeout}}{20}$. This is a good result for a verified and generated software, especially as SPIN builds a tailored checker for each model, whereas CAVA's is general.

Further, we tested multiple properties on scaling versions of the leader election protocol and the "Dining Philosophers". Here, the LTL-to-Büchi translation is important. As of this time, the implementation in CAVA is tailored to verification, not efficiency. This leads to larger-than-necessary state spaces, in particular for formulas containing U. Therefore the slowdown is a factor between 9 and 70. For negative properties, SPIN found 75 of 77 counter-examples in less than 10 seconds, CAVA 70 of 77.

The main reason for the difference in performance is the lack of destructive updates in a purely functional program. In particular we must use trees as our main data structure, yielding a logarithmic overhead. Arrays can only be used when updates are seldom, as they cannot be updated in-place but need to be copied in full. Moreover we cannot utilize pointers for keeping a reference to a changing structure, but have to look up information each time. The consequences are shown in Fig. 5b. We ran "Dining Philosophers" modeled in two different ways: using three arrays of length N, and using $3N$ different variables. The amount of variables has a very notable impact on performance, even though this does not influence the state-space.

6 Future Work

In the previous sections we outlined the current state of the Promela implementation for our model checker CAVA. As already indicated throughout the paper, there still are additional targets which are to be addressed.

In Section 2 we mentioned different parts of Promela which are not implemented yet. For those where it is possible, we strive to add support. This also includes overflowing for integer types, as there may be valid use cases.

Furthermore, the current formalization should see further separation between abstraction and implementation, as was done in the other parts of CAVA. This also allows for an even better presentation of the semantics of Promela for one, and, due to possible refinement, additionally clears the way for implementing optimizations without changing semantics.

As already hinted in the previous section, there are several opportunities for performance enhancements. For faster lookups, we already employ hashing. Here, a theory of consistent hashes is planned, to introduce a technique eliminating the rehashing of unchanged structures.

Further future work includes the introduction of new algorithms for emptiness detection, yielding an even better performance.

An important topic to work on are the additional non-trivial optimizations of SPIN, this includes partial-order reduction [14]. This technique is an important optimization used in SPIN to drastically reduce the size of the state-space. This technique needs to be formalized in Isabelle/HOL and then be integrated into CAVA.

Acknowledgements We are very grateful for the help and input of Javier Esparza, Dennis Kraft, Peter Lammich, Andreas Lochbihler, Philipp Meyer, and Tobias Nipkow.

References

1. Promela database. `http://www.albertolluch.com/research/promelamodels`, accessed: 2013-01-15
2. Promela manual pages. `http://spinroot.com/spin/Man/promela.html`, accessed: 2013-02-07
3. Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M.: Memory-efficient algorithms for the verification of temporal properties. Formal Methods in System Design 1(2/3), 275–288 (1992)
4. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.G.: A fully verified executable LTL model checker. In: Sharygina, N., Veith, H. (eds.) CAV. LNCS, vol. 8044, pp. 463–478. Springer (2013)
5. Gallardo, M.d.M., Merino, P., Pimentel, E.: A generalized semantics of PROMELA for abstract model checking. Formal Aspects of Computing 16(3), 166–193 (2004)
6. Gava, F., Fortin, J., Guedj, M.: Deductive verification of state-space algorithms. In: Johnsen, E.B., Petre, L. (eds.) IFM. LNCS, vol. 7940, pp. 124–138. Springer (2013)
7. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Dembinski, P., Sredniawa, M. (eds.) Proc. Int. Symp. Protocol Specification, Testing, and Verification. IFIP Conference Proceedings, vol. 38, pp. 3–18. Chapman & Hall (1996)
8. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS. LNCS, vol. 6009, pp. 103–117. Springer (2010)

9. Holzmann, G.J.: The Spin Model Checker — Primer and Reference Manual. Addison-Wesley (2003)
10. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an OS kernel. In: Matthews, J.N., Anderson, T.E. (eds.) Proc. ACM Symp. Operating Systems Principles. pp. 207–220. ACM (2009)
11. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft's algorithm. In: Beringer, L., Felty, A. (eds.) ITP. LNCS, vol. 7406, pp. 166–182. Springer (2012)
12. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
13. Pelánek, R.: BEEM: Benchmarks for explicit model checkers. In: Bošnački, D., Edelkamp, S. (eds.) Model Checking Software. LNCS, vol. 4595, pp. 263–267. Springer (2007)
14. Peled, D.: Combining partial order reductions with on-the-fly model-checking. In: Dill, D.L. (ed.) CAV, LNCS, vol. 818, pp. 377–390. Springer (1994)
15. Pollack, R.: How to believe a machine-checked proof. In: Sambin, G., Smith, J. (eds.) Twenty Five Years of Constructive Type Theory. Oxford Univ. Press (1998)
16. Schwoon, S., Esparza, J.: A note on on-the-fly verification algorithms. In: Halbwachs, N., Zuck, L. (eds.) TACAS. LNCS, vol. 3440, pp. 174–190. Springer (2005)
17. Shankar, N.: Trust and automation in verification tools. In: Cha, S., Choi, J.Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA. LNCS, vol. 5311, pp. 4–17. Springer (2008)
18. Sharma, A.: A refinement calculus for Promela. In: ICECCS. pp. 75–84. IEEE (2013)
19. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: LICS. pp. 332–344. IEEE Computer Society (1986)
20. Weise, C.: An incremental formal semantics for PROMELA. In: Proceedings of the 3rd International SPIN Workshop (1997)