

Model Checking CoreASM Specifications

Roohbeh Farahbod, Uwe Glässer and George Ma

Software Technology Lab
Simon Fraser University
Burnaby, B.C., Canada
{rfarahbo, glaesser, gzm}@cs.sfu.ca

Abstract. In this paper we present an approach to model checking abstract state machines using the Spin model checker. We give an algorithm for automatically transforming ASM specifications written in CoreASM [1] into Promela specifications. Though an algorithm for translating ASMs into Promela has already been presented in [2], our method supports a more powerful ASM language, including support for n-ary functions and extended rule forms. Specifically, our translation also supports distributed abstract state machines.

1 Introduction

As the adoption of information technology continues to grow, with software pervading many aspects of modern life, the importance of engineering software efficiently and correctly is paramount. Designing high quality software is extremely challenging, and software failures can be very costly. These problems have motivated the use of *formal methods* in software engineering which allows for catching and fixing design errors and inconsistencies early in the design process when the cost of making changes is much higher.

While the rigor and structure imposed by using formal methods can help improve the quality of software by removing ambiguities and sharpening understanding of system requirements, software specifications are still ultimately descriptions of algorithms. Formal verification techniques aim to increase software quality by providing a consistent logical framework in which to reason about the correctness of programs. Within such a framework, a property of a system can actually be *proven* using model checking techniques.

In this paper we present an approach to model checking abstract state machines (ASMs). The effectiveness of using abstract state machines as a formal method has been proven through their application in industrial settings, as illustrated in works by Börger, Gurevich, Glässer, and others [3–5]. We believe that applying model checking to ASMs can be a useful tool in assuring the correctness and quality of software specifications. This work is done in the context of a larger project called CoreASM [1, 6], which aims to provide a comprehensive and extensible tool environment for the design and validation of systems using the ASM formalism. The CoreASM engine facilitates experimental validation of ASM models by supporting execution of ASM specifications. However, experimental

validation without model checking cannot formally verify the correctness of a system with respect to all of its possible behaviors. Hence, the goal of this work is to provide model checking support for CoreASM specifications by translating CoreASM models into Promela models, which can be verified by the Spin model checker.¹ We present a novel approach to performing this transformation so as to support distributed abstract state machines. Moreover, we aim to provide a tool that is simple to use and well integrated with the other existing CoreASM tools. This work also illustrates the extensibility of CoreASM by presenting specifications to CoreASM plug-ins which allow function signatures and correctness properties to be included as part of a specification.

2 Related Work

This work is certainly not the first of its kind. Del Castillo and Winter present an approach for model checking abstract state machines in [8]. In this work, specifications written in ASM-SL, the ASM language for the ASM Workbench tool [9], are translated into input for the SMV model checker. Gargantini and Riccobenne in [10] present a method for model checking ASMs by translating specifications written for the AsmGofer tool [11] into Promela. In a more recent work, Tang and Ternovska present a method for bounded model checking of ASMs using Answer Set Programming [12]. Martin Kardos has also recently developed a model checker for the Abstract State Machine Language (AsmL)² [13].

The work presented here is similar to the work done by Gargantini and Riccobenne, as our work also uses Spin to model check ASMs. However, the translation procedure described here extends their work in several aspects, such as a) support for all basic ASM rules, save for **import**; b) support for arbitrary n -ary functions; and c) support for distributed abstract state machines.

3 Extending CoreASM for Model Checking

In order to provide model checking support for CoreASM, we utilize the CoreASM engine to translate CoreASM models into equivalent Promela models which can be verified using the Spin model checker [14]. From a high level perspective, the steps in the translation and verification process are as follows (see also Fig. 1):

1. A CoreASM specification is loaded and parsed by the CoreASM engine, producing an Abstract Syntax Tree.
2. The Abstract Syntax Tree is translated into Promela.
3. Spin is invoked to generate a verifier of the Promela model, producing C code.

¹ Spin is a widely used automata based model checker that has been used extensively in the design of asynchronous distributed systems [7].

² AsmL was developed by the Foundations of Software Engineering (FSE) group at Microsoft Research (see <http://research.microsoft.com/fse/asml/>).

4. The C code is compiled, generating a custom verifier of the CoreASM specification.
5. The verifier is run, producing a counter example if the property being checked does not hold.

Using the abstract syntax tree as the basis for translation allows for a structured translation scheme and the straight-forward application of a recursive translation procedure.

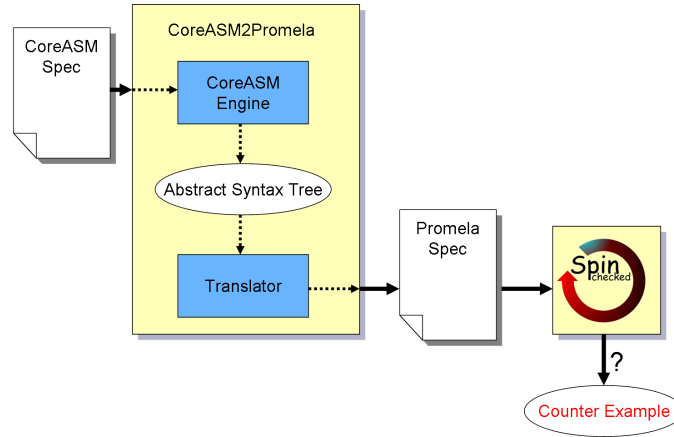


Fig. 1. CoreASM2Promela: Overall Verification Process

In order to properly translate CoreASM specifications into Promela models, we needed to extend the CoreASM language by two new plug-ins, namely the *Signature Plug-in* and the *Property Plug-in*, to support declaration of function signatures and specification of LTL properties as part of CoreASM specifications. The following sections briefly describe these two plug-ins. A comprehensive specification of these plug-ins is provided in [15].

3.1 Signature Plug-in

In principle, CoreASM functions are untyped alike ASM functions. While this is desirable in initial specification phases focusing on exploring the problem space, domain and range types of functions often add useful semantic information to a refined specification, for instance, to improve its understandability, to implement runtime type checking, and also to facilitate model checking. The Signature Plug-in extends the grammar of the CoreASM language with its own syntactic patterns to provide means to declare functions, universes, and enumerated backgrounds directly in a CoreASM specification thereby adding type information to CoreASM.

To declare functions, the Signature plug-in extends the CoreASM language with the following syntactic patterns, which can appear in the header of a specification:

function *function-class* $f : domain_1 * \dots * domain_n \rightarrow range$ [**initially** *value*]

where *function-class* is either of keywords **controlled**, **static**, or **monitored**. The initialization expression (*value*) may be a basic expression, for nullary functions, or a map expression, for n -ary functions. Before the function is created, the expression giving its initial value is evaluated.

To declare universes, the Signature plug-in provides the following patterns:

universe *universe* [= { e_1, \dots, e_n }]

The above pattern allows the specification writer to declare a universe along with an optional set of named initial member elements. Of course, a declared universe can still be extended using standard methods, namely by using the **extend** rule, which imports a new element to a universe, or by setting the value of the corresponding universe membership predicate to *true* for a given element.

The Signature plug-in also provides a similar pattern to declare enumerated backgrounds:

enum *enum-background* = { e_1, \dots, e_n }

3.2 Property Plug-in

The Property Plug-in is a small plug-in that allows correctness properties, expressed as LTL formulas, to be included in the header of a CoreASM specification. Presently, specified properties do not have any meaning during ASM simulations (although it may be possible to extend the Property plug-in to check simple global assertions). Correctness properties are only applicable during model checking, and are translated by our CoreASM to Promela translator.

The Property plug-in provides the following pattern to declare new LTL properties:

[**check**] **property** *LTL-property*

Including the keyword **check** with a property declaration indicates that the property should be checked during model checking.

The Property plug-in improves the usability of the Spin model checker, since Spin does not allow LTL properties to be included directly in a specification. In Spin, properties are defined by describing the behavior of a property automaton. Moreover, Spin only allows a single property automaton in each model, while the Property plug-in allows multiple properties to be specified for a single specification.

4 From CoreASM to Promela

This section gives an overview of our procedure for transforming a CoreASM specification into a Promela model. We refer the reader to [15] for details.

4.1 States

As Spin can only check finite models, the translation scheme is limited to CoreASM specifications which have finite states. Thus, the translation supports only static universes and enumerated backgrounds. Each element from a given universe is mapped to an integer value, starting from zero, and is defined as a Promela constant.

Functions are the core part of the state. Our translation supports *controlled*, *static*, and *monitored* functions. CoreASM functions are translated as Promela global variables. Nullary functions are translated as basic integer variables. N -ary functions are translated as multi-dimensional arrays, in which each array element corresponds to a unique function location. Although Promela does not directly support multi-dimensional arrays, they can be created indirectly by chaining typedef statements.

For each controlled function, two variables are declared: one representing the function in the current state and the other one representing the intended value for the next state. Updates only affect the next-state variable.

Monitored functions are updated in between each ASM step. The next value of a monitored function is chosen non-deterministically from the values in its range, using a Promela selection statement. If g is a monitored function and the values v_1, \dots, v_n are in its range, g is updated by the following statement:

```
inline monitoredUpdate() {
  if
  :: g = v_1;
  :: g = v_2;
  ...
  :: g = v_n;
  fi;
  ...
}
```

If g is has arity greater than zero, each of its locations is updated in the same fashion as what is shown above.

The translation also supports all operators that can be mapped to native operators in Promela, as well as universal and existential quantification expressions.

4.2 Rules

All basic ASM rules, including conditional, block, forall, and choose rules, are supported by the translation, except for those which introduce new elements from the reserve, such as **import** and **extend**, since these rules can potentially produce models with infinite state space. The translation also supports macro rules by translating them as Promela inline procedures.

Example: Choose Rule A **choose** rule with the following syntax:

```
choose id in value with guard do rule1 ifnone rule2
                    optional                                optional
```

is translated as a non-deterministic Promela conditional statement. For each $v_i \in \text{value}$, all occurrences of id in $guard$ and $rule_1$ are replaced by v_i . Each of the new rules that result from the substitution becomes a branch of the main conditional statement. In the code below, the expressions “guard[v/id]” and “rule₁[v/id]” denote $guard$ and $rule_1$ with all instances of the identifier id replaced with value v . If an **ifnone** clause is provided, the associated rule ($rule_2$) becomes the default action.

```
if
:: guard[v_1/id] -> rule_1[v_1/id];
:: guard[v_2/id] -> rule_1[v_2/id];
...
:: guard[v_n/id] -> rule_1[v_n/id];
:: else -> rule_2;
fi;
```

Program Rules Rules which are agent programs are handled specially; they are translated as Promela proctype declarations, which define new process types. The special function *self* is defined locally in each process. When a process is instantiated, it is given a unique value for *self*, namely the value from the translation of Agent universe declaration. In each agent step, monitored functions are updated before the actual program rule is executed. After the program rule is executed, controlled functions are updated, thereby performing the agent’s move and producing the next ASM state.

```
proctype programRuleName(byte self) {
do
:: atomic {
monitoredUpdate();
/* Rule Body Translation */
...
functionUpdate();
};
od;
};
```

4.3 DASM Simulation Model

Every Promela model has a special initial process (which is specified using the keyword *init*) that is the first process run by Spin. In our translation from CoreASM to Promela we use the *init* process to initialize the state of the simulated ASM.

In the ASM initialization section of the *init* process, all locations are first given the value *undef*. Then, if the function declaration section of a specification

includes initial values, these initial values are set. Afterwards, the actual initial rule given by the specification writer is executed. Processes corresponding to the program of each of the agents are then instantiated, with the value of each process' *self* argument set to identify each DASM agent. Note that it is possible for multiple agents to share the same process type.

```

init {
  atomic {
    functionInit ();
    InitRule ();
    functionUpdate ();
    run program1(agent1);
    ...
    run programN(agentN);
  };
}

```

Spin's process scheduler is non-deterministic and as we have program rules surrounded by atomic blocks, during model checking every possible interleaved sequence of process executions is considered. Thus, every possible sequence of DASM agent moves is considered. Interleaving semantics model the partially ordered runs of distributed abstract state machines faithfully, since the coherence condition implies that all linearizations of a partially ordered run result in the same final state.

5 FLASH Cache Coherence Protocol Case Study

The FLASH Cache Coherence Protocol coordinates the sharing of memory among the processing nodes of the Stanford FLASH multiprocessor. Winter [16] uses the protocol as a model checking case study in her PhD thesis on ASM2SMV. We compare the results of using our tool against ASM2SMV³. In the FLASH multiprocessor, distributed memory is partitioned into lines and each line is associated with a home-node which hosts the part of the physical memory where the line resides. The sharing of memory is facilitated by holding local copies of data at each node. The Cache Coherence Protocol guarantees that none of the nodes hold a copy of data that is out of date.

In these experiments, we use an erroneous specification of the protocol, based on the original model Winter used and then corrected, to elicit counterexamples from the model checkers, in addition to verifying a true property. The following properties were tested:

- P2: No two nodes have exclusive access to the same line at any time.
 $\forall i \forall j \neq j' \mathbf{G}(\neg(State(node_j, line_i) = exclusive \wedge State(node_{j'}, line_i) = exclusive))$

³ Tests were run on a Sun machine with a 1.2 GHz UltraSparc processor and 4 GB of main memory, using Spin, version 4.2.8, and NuSMV, version 2.4.1.

- P3: Every request will eventually be acknowledged.
 $\forall i \forall j \mathbf{G}(CurPhase(node_j, line_i) = wait \rightarrow \mathbf{F}(CurPhase(node_j, line_i) = ready))$
- P4: Whenever a node obtains shared access to a line, it will be marked as a sharer of the line.
 $\forall i \forall j \mathbf{G}((State(node_j, line_i) = shared \rightarrow \mathbf{X}(Sharer(line_i, node_j) = true)) \vee (Sharer(line_i, node_j) = true \rightarrow \mathbf{X}(State(node_j, line_i) = shared)))$

P2 and P3 are not satisfied by the model, while P4 holds. In our experiments, we varied three parameters: N the number of nodes, L the number of lines, and Q the size of the message queue at each node. Also, since P2 and P4 do not require scheduling fairness, we performed the tests both with and without explicit round robin scheduling for the sake of comparison. The results of the experiments are shown in Table 1.⁴

Parameters	Property	CoreASM2Promela	CoreASM2Promela Round Robin	ASM2SMV (BDD)
N=2, L=1, Q=1	P2 P3 P4	6s N/A 196s	43s 7s 1,894s	438s 921s 76s
N=2, L=2, Q=1	P2 P3 P4	85s N/A 5,187s	5,376s 671s 188,907s	MEM MEM MEM
N=3, L=1, Q=2	P2 P3 P4	164s N/A MEM	16,356s 398s MEM	MEM MEM MEM

Table 1. Flash Cache Coherence Protocol Model Checking Results

6 Conclusion

In this paper we introduce an approach to model checking ASM specifications based on the CoreASM modeling framework. We have extended the CoreASM engine to add support for function signatures and LTL properties, and we have developed a new tool to translate CoreASM specifications into Promela models that can be model checked with Spin.

We have used our tool to model check several non-trivial ASM specifications and compared our results to those produced using Winter’s ASM2SMV tool. The results show that counterexamples to false properties are found more quickly using our translation tool and Spin, while true properties are verified more quickly using ASM2SMV. In addition, our tool is able to translate larger specifications

⁴ A MEM entry in the table indicates that there was not enough memory to complete the verification.

on which ASM2SMV fails. There were also a number of cases where properties that we checked against our Promela models could not be checked on the resultant SMV models due to insufficient memory. The details of the case studies and a comprehensive discussion of the results are presented in [15].

Acknowledgements Our sincere appreciation to the anonymous reviewers of this paper for their valuable comments and suggestions for improvements.

References

1. CoreASM Development Team: (The CoreASM Project) <http://www.coreasm.org>.
2. Gargantini, A., Riccobene, E., Rinzivillo, S.: Using Spin to generate tests from ASM specifications. In: Abstract State Machines 2003, Springer (2003) 263–277
3. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer-Verlag (2003)
4. Glässer, U., Gotzhein, R., Prinz, A.: The formal semantics of SDL-2000: status and perspectives. *Comput. Networks* **42** (2003) 343–358
5. Börger, E., Päppinghaus, P., Schmid, J.: Report on a Practical Application of ASMs in Software Design. In Y. Gurevich and P. Kutter and M. Odersky and L. Thiele, ed.: Abstract State Machines: Theory and Applications. Volume 1912 of LNCS., Springer-Verlag (2000) 361–366
6. Farahbod, R., Gervasi, V., Glässer, U.: CoreASM: An extensible ASM execution engine. *Fundamenta Informaticae* (2007) 71–103
7. Holzmann, G.J.: The model checker SPIN. *Software Engineering* **23** (1997) 279–295
8. Del Castillo, G., Winter, K.: Model Checking Support for the ASM High-Level Language. In Graf, S., Schwartzbach, M., eds.: Proceedings of the 6th International Conference TACAS 2000. Volume 1785 of LNCS., Springer-Verlag (2000) 331–346
9. Del Castillo, G.: Towards Comprehensive Tool Support for Abstract State Machines. In Hutter, D., Stephan, W., Traverso, P., Ullmann, M., eds.: Applied Formal Methods — FM-Trends 98. Volume 1641 of LNCS., Springer-Verlag (1999) 311–325
10. Gargantini, A., Riccobene, E.: ASM-based Testing: Coverage Criteria and Automatic Test Sequence Generation. *Journal of Universal Computer Science* **7** (2001) 1050–1067
11. Schmid, J.: (Executing ASM Specifications with AsmGofer) Last visited Sep. 2005, www.tydo.de/AsmGofer/.
12. Tang, C.K.F., Ternovska, E.: Model checking abstract state machines with answer set programming. In: LPAR. (2005) 443–458
13. Kardos, M.: An approach to model checking asml specifications. In: Proceedings of the 12th International Workshop on Abstract State Machines. (2005) 289–304
14. Holzmann, G.: The Spin Model Checker, Primer and Reference Manual. Addison-Wesley, Reading, Massachusetts (2003)
15. Ma, G.Z.: Model checking support for CoreASM: Model checking distributed abstract state machines using spin. Master’s thesis, Simon Fraser University, Burnaby, Canada (2007)
16. Winter, K.: Model Checking Abstract State Machines. PhD thesis, Technical University of Berlin, Germany (2001)