Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 190 (2007) 85–97
www.elsevier.com/locate/entcs

Automated Verification of Completeness and Consistency of Abstract State Machine Specifications using a SAT Solver

Martin Ouimet[1] Kristina Lundqvist[2]

Embedded Systems Laboratory
Massachusetts Institute of Technology
Cambridge, MA, USA 02139

Abstract

In the requirements engineering community, consistency and completeness have been identified as important properties of system specifications. Custom algorithms to verify these properties automatically have been devised for a number of specification languages, including SCR, RSML, and Statecharts. In this paper, we provide means to automatically verify completeness and consistency of Abstract State Machine (ASM) specifications. The verification is performed using a widely available tool, a *SAT* solver. The use of a *SAT* solver removes the need for designing and fine tuning language specific verification algorithms. Furthermore, the use of a *SAT* solver automates the verification procedure and produces a counterexample automatically when a specification is incomplete or inconsistent. We provide an algorithm to translate ASM specifications to a *SAT* problem instance. The translation is illustrated using the TASM toolset in conjunction with the "production cell system" case study.

Keywords: Verification, Completeness, Consistency, SAT, Abstract State Machines

1 Introduction

Consistency and completeness were identified as useful properties of specifications in [8] and in [9]. In the context of specification of embedded systems, *completeness* of the specification is defined as the specification having a response for every possible class of inputs. In the same context, *consistency* is defined as the specification being free of contradictory behavior, including unintentional non-determinism [9]. Formal definitions of these properties, in the context of Abstract State Machine (ASM) specifications, are given in Section 3. Traditionally, verifying these properties was accomplished manually by system specifiers, through inspection of specifications. Because a specification is likely to evolve during the engineering lifecycle,

[1] Email: mouimet@mit.edu
[2] Email: kristina@mit.edu

doi:10.1016/j.entcs.2007.08.008

the ability to verify these properties automatically can ease and shorten the analysis process [9]. Language specific verification algorithms have been proposed in [8] and in [9]. In contrast, the approach proposed in this paper is not language specific and can be reused for other languages. The proposed approach achieves verification by translating specifications to formulas in propositional logic, formulating completeness and consistency as a boolean satisfiability problem (SAT) [20], and automating the verification procedure by using a generally available solver.

Abstract State Machines (ASM) have been used to specify, analyze, and verify hardware and software systems at different levels of abstraction [1]. Abstract State Machines have also been used to automate engineering activities, including verification using model checkers [21] and test case generation [5]. The Timed Abstract State Machine (TASM) language is an extension of the ASM language that includes facilities for specifying non-functional properties, namely time and resource consumption [16]. The TASM language and its associated toolset have been used to model and simulate real-time systems [13], [14], and [15]. The relationship between ASM and TASM is quite close and the notions of completeness and consistency introduced in this paper are equally applicable to both ASM and TASM.

In this paper, an approach to automatically verify consistency and completeness of TASM specifications is provided. The verification is achieved by mapping TASM specifications to boolean formulas in Conjunctive Normal Form (CNF). The specified mapping is derived using the structural properties of the specification and does not require the generation of a global reachability graph, thereby avoiding the infamous state space explosion problem [9]. The proposed mapping could also be applied to ASM specifications because the mapping does not consider the extensions of the TASM language. The mapping to boolean formulas in CNF allows for automated verification using a SAT solver. The mapping is achieved in such a way that consistency and completeness are expressed as unsatisfiability of the boolean formulas. If the TASM specification is incomplete or inconsistent, the SAT solver will generate an assignment which makes the boolean formula satisfiable. This assignment serves as the counterexample to exercise the incompleteness or inconsistency of the specification.

2 Related Work

The definition and automated verification of completeness and consistency of specifications were introduced in [8] and in [9]. In [8], the RSML language, a hierarchical state-based language, is used to express requirements. The language is automatically analyzed for completeness and consistency using an algorithm specifically targeted for the RSML language. In [9], a similar approach is used for analysis of requirements expressed in the SCR language. These two approaches rely on specific purpose algorithms for the efficient and automated analysis of consistency and completeness. Consequently, the proposed algorithms cannot be reused for other languages. In contrast, the approach proposed in this work utilizes a general purpose solver, a SAT solver. The proposed translation from TASM specifications to

boolean formulas in CNF can be reused for other specification languages. The use of a *SAT* solver guarantees that the analysis procedure is optimized.

In the ASM community, various derivatives of the ASM language have been developed, including the ASM Workbench [3] and the Abstract State Machine Language (AsmL) [7]. A mapping between the ASM Workbench language (ASM-SL) and finite state machines, for the purpose of model checking, was proposed in [21]. A mapping between the AsmL language and finite state machines was proposed in [5]. The mapping to finite state machines was used for automated test case generation [6]. The mapping proposed in this paper resembles the mappings proposed in these two approaches except that it ignores the effect of rule applications and does not need to generate a global reachability graph. The proposed mapping concerns itself only with relationships between rule guards inside a single machine and hence produces a smaller state space than might be generated through a complete reachability graph.

SAT solvers have been used for a variety of automated analysis, including test case generation [10], [18]. Although the *SAT* problem is known to be NP-Complete, the use of *SAT* solvers has been shown to be useful in a wide range of cases. *SAT* solvers and model checkers show similarities in their benefits, namely automation of the verification procedure and automation of the counterexample generation. *SAT* solvers and model checkers also show similarities in their drawbacks, namely the potential for state space explosion and the resulting intractability of large state space exploration.

3 Definitions

Abstract State Machines (ASM) is a formal language and an associated methodology for system design and analysis. The language and methodology have been used to model a variety of hardware and software systems, at different levels of abstraction [2].There have been a number of derivatives of the ASM language, including the ASM Workbench language (ASM-SL) [3], the Abstract State Machine Language (AsmL) [7], and the Timed Abstract State Machine (TASM) language [16]. While these derivatives have syntactic and semantic differences, they all rely on the basic concepts of ASMs. The abstract state machine formalism revolves around the concepts of an abstract machine and an abstract state. System behavior is specified as the computing steps of the abstract machine. A computing *step* is the atomic unit of computation, defined as a set of parallel updates made to global *state*. A *state* is defined as the values of all variables at a specific step. A machine *executes* a step by yielding a set of state updates. A *run*, potentially infinite, is a sequence of steps. The structure of an ASM is a finite set of rules, written in precondition-effect style. For an ASM that contains n rules, a *block* machine, also called a machine in *canonical* form has the following structure:

$$\begin{aligned} R_1 &\equiv if\ G_1\ then\ E_1 \\ R_2 &\equiv if\ G_2\ then\ E_2 \\ &\vdots \\ R_n &\equiv if\ G_n\ then\ E_n \end{aligned} \tag{1}$$

The guard G_i is the condition that needs to be enabled for the effect of the rule, E_i, to be applied to the environment. The effect of the rule is grouped into an *update set*, which is applied atomically to the environment at each computation step of the machine. For a complete description of the theory of abstract state machines, the reader is referred to [2].

3.1 The Timed Abstract State Machine (TASM) Language

The Timed Abstract State Machine (TASM) language [16] is an extension of the ASM language for the specification and analysis of real-time systems. The TASM language extends the specification of rules by enabling the specification non-functional properties, namely time and resource consumption. The semantics of rule execution extend the update set concept by including the duration of the rule execution and a set of resource consumptions during the rule execution.

In the TASM language, the canonical form given in equation 1 remains the same, except for the effect expressions. In the TASM language, the effect expressions, E_i, are extended to reflect the time and resource consumption specification. Other features of the TASM language include hierarchical and parallel composition, expressed through the use of *sub* machines and *function* machines. The definition of consistency and completeness, in terms of TASM specifications, is expressed as relational properties between the rule guards G_i. Consequently, the definitions of consistency and completeness given in Section 3.3 and Section 3.4, as well as the verification approach, are equally applicable to the TASM language and other derivatives of the ASM language. The approach could also be applicable to other languages as well, as long as these languages can be expressed in the canonical form given in equation 1. For a complete description of the TASM language, the reader is referred to [12].

3.2 The Satisfiability Problem

The satisfiability problem, also known as *SAT* for short, is the archetypical *NP-Complete* problem in the theory of computation [20]. The problem involves determining whether a boolean formula is satisfiable. A *boolean formula* is composed of a set of atomic propositions and operations. Atomic *propositions* are boolean variables that can take the values *TRUE* or *FALSE*. The propositions are connected using parentheses and the operations *NOT*, *AND*, and *OR*, represented by the symbols $\neg$, $\wedge$, and $\vee$. A boolean formula is *satisfiable* if there is an assignment of values to propositions which makes the formula *TRUE*. If no such assignment exists, the formula is *unsatisfiable*. A sample *SAT* problem is shown below:

$$(b_1 \ \vee \ b_2) \ \wedge \ (b_1 \ \vee \ b_3)$$

3.3 Completeness

Informally, *completeness* is defined as the specification having a response for every possible input combination. In the TASM world, for a given machine, this criteria means that a rule will be enabled for every possible combination of its monitored variables. The *monitored* variables are the variables in the environment which affect the machine execution. Formally, the disjunction of the rule guards of a given machine must form a tautology. The letter S is used to denote an instance of the SAT problem. The completeness problem can be expressed as a SAT problem in the following way:

For n rules:

$$S \ \equiv \ \neg \ (\ G_1 \ \vee \ G_2 \ \vee \ \ldots \ \vee \ G_n \ )$$

$$ASM = \begin{cases} \text{complete} & \text{if } S \text{ not satisfiable} \\ \text{incomplete} & \text{if } S \text{ satisfiable} \end{cases}$$

The completeness problem is casted as the negation of the disjunction so that counterexamples can be generated by the SAT solver. If S is satisfiable, all the assignments that make S satisfiable can be automatically generated by the SAT solver. If S is not satisfiable, the specification is complete.

3.4 Consistency

Informally, for a state-based specification, *consistency* is defined as no state having more than one transition enabled at the same time [8]. The definition given in [9] is similar but extended to include other properties of the specification such as syntactical correctness and type checking. The definition of consistency adopted in this approach is the same as in [8]. In terms of TASM specifications, this definition states that no two rules can be enabled at the same time. This definition will lead to a set of SAT problems to define consistency:

For each pair of rules R_i, R_j where $1 \leq i < j \leq n$:

$$S \ \equiv \ G_i \ \wedge \ G_j$$

$$ASM = \begin{cases} \text{consistent} & \text{if } S \text{ not satisfiable} \\ \text{inconsistent} & \text{if } S \text{ satisfiable} \end{cases}$$

This definition yields a set of $\binom{n}{2}$ SAT problems. The individual SAT problems can also be composed into a single SAT problem. As for completeness, the SAT problem is defined in such a way that if the specification is not consistent, a counterexample is automatically generated. If S is satisfiable, all the assignments that make S satisfiable can be automatically generated by the SAT solver.

4 Translation to SAT

The TASM language is a typed language that includes integer datatypes, boolean datatypes, and user-defined types. User-defined types are analogous to enumeration types in programming languages. The TASM language is a subset of the ASM language and does not include all of the constructs of the ASM language. For example, the *choose* construct is not part of TASM. The concepts from the ASM language included in the TASM language are the same as defined in [21]. The translation from TASM to SAT involves mapping the rule guards, G_i, to boolean propositions, b_i, in Conjunctive Normal Form (CNF). The following subsections explain how this translation is performed.

4.1 Boolean and User-Defined Datatypes

In the TASM language, user-defined datatypes and boolean datatypes are simple types that can take values for a finite set. Boolean variables can take one of two values (*true* or *false*). User-defined types can take one of multiple values, as defined by the user. In typical specifications, user-defined types rarely exceed five or six members.

The only operations defined for boolean and user-defined datatypes are the comparison operators, = and ! =. No other operator is allowed for boolean and user-defined datatypes. In the translation to SAT, we take the equality operator (=) to mean a non-negated proposition (e.g., b_1). The operator ! = is translated to mean a negated proposition (e.g., $\neg b_1$). The translation to SAT for these datatypes involves 2 steps. The first step is generating the *at least one* clause and the *at most one* clause for each variable of type boolean or of type user-defined type. The second step involves formulating the property to be verified as a clause in CNF, S, according to the definitions in Section 3. The *at least one* clause ensures that the variable must take at least one value from its finite set. This clause is simply the disjunction of equality propositions for each possible value that the variable can take. The *at most one clause* is a clause that ensures that each variable can take at most one value from its finite set.

To illustrate the generation of the *at least one* and *at most one* clauses, the following type is introduced: $type1 := \{val_1, val_2, \ldots, val_n\}$. A variable of type boolean can be viewed as a variable of type $type1$ where n = 2. First, the set of propositions is generated. In SAT, a proposition is a single letter with a subscript (e.g., b_i). For a variable named *var* of type $type1$, the following propositions would

be generated, where the b_i's represent the SAT atomic propositions and the right hand side represents the meaning of the proposition in the TASM context:

$$\begin{aligned} b_1 &: \ var \ = \ val_1 \\ b_2 &: \ var \ = \ val_2 \\ &\vdots \\ b_n &: \ var \ = \ val_n \end{aligned}$$

The *at least one* clause, denoted C_1 for this variable would be:

$$C_1 \ \equiv \ b_1 \ \vee \ b_2 \ \ldots \ \vee \ b_n$$

The *at least one* clause ensures that at least one of the b_i's must be true for the clause to be true. The *at most one* clause ensures that no two b_i's can be true at the same time. The *at most one clause*, denoted C_2 is the conjunction of multiple clauses:

$$\begin{aligned} C_2 \ \equiv & (\neg b_1 \ \vee \ \neg b_2 \ \ldots \ \vee \ \neg b_n) && \wedge \\ & (b_1 \ \vee \ \neg b_2 \ \ldots \ \vee \ \neg b_n) && \wedge \\ & (\neg b_1 \ \vee \ b_2 \ \ldots \ \vee \ \neg b_n) && \wedge \\ & \qquad \vdots && \wedge \\ & (\neg b_1 \ \vee \ \neg b_2 \ \ldots \ \vee \ b_n) \end{aligned}$$

The at most one clause generates $n + 1$ clauses, one for the full negations of the propositions and one for each $n - 1$ negations of propositions. This combination ensures that at most one of the clauses can be true. The conjunction $C_1 \wedge C_2$, which is already in conjunctive normal form, serves to enforce the "exactly one value per variable" constraint, also called *type enforcement*. The rule guards are made up of propositions that already exist in the proposition catalog. For each rule guard in the problem formulation S, for each constraint in the guards, if the constraint is of the form $var = val_i$, its corresponding proposition b_i is looked up in the catalog and substituted in the problem formulation S. If, on the other hand, the constraint is of the form $var \ != val_i$, the b_i corresponding to $var = val_i$ is looked up in the proposition table and the constraint in the guard is substituted by its negation, $\neg b_i$. Once the substitution is done in the rule guards, the formulated problem S is then converted to conjunctive normal form using the well-known algorithm in [20]. The result of this substitution and conversion to CNF yields S with only atomic boolean propositions. The full SAT problem can then be formed by the conjunction of S, C_1, and C_2:

$$\text{Full } SAT \text{ problem } \equiv S \wedge C_1 \wedge C_2$$

4.2 Integer Datatypes

Similarly to boolean datatypes and user-defined datatypes, integer datatypes take values from a finite set. However, the number of values that integers can take is much larger than for boolean datatypes and much larger than for typical user-defined types. For example, in the TASM language, integers range from -32,768 to 32,767. While the approach suggested above for boolean and user-defined types might also work for integer types, the enumeration of all 65,536 possible values would be intractable for a single integer variable. The adopted mapping for integer variables relies on the fact that even though integers are used in TASM specifications, they are used in such a way that they could be replaced by user-defined types. In other words, in TASM specifications, the full range of integers is typically not used.

Nevertheless, integer datatypes are more complex than boolean and user-defined types because more operations are defined for integer datatypes. These operations are comparison operators and arithmetic operators. The comparison operators are $=$, $!=$, $<$, $<=$, $>$, and $>=$. The arithmetic operators are $+$, $-$, $*$, and $/$. For the suggested translation, constraints on integer variables must be of the form $< var > < com\_pop > < expr >$, where $< var >$ is an integer variable $< comp\_op >$ is a comparison operator and $< expr >$ is an arbitrary arithmetic expression that can contain constants, variable references, function machine calls, and operators. The restriction is that the left hand side of constraints can contain only a variable, with no arithmetic expressions allowed. The translation proposed in this section, deals only with linear constraints whose right hand sides are constants. Arbitrary symbolic right hand sides will be handled in future work, as explained in section 6.

The key idea behind the translation is to convert each integer variable to a user-defined type. This is achieved by collecting all of the constraints on a given integer variable and extracting the intervals that are of interest. These intervals become the members of the user-defined types. Once the integer type has been converted to a user-defined type in this fashion, it can then be converted to a boolean formula using the approach from Section 4.1. The algorithm to reduce integer variable to user-defined types consists of 4 steps. For each monitored variable of type integer:

(i) Collect all constraints on the variable from S

(ii) Sort all constraints in ascending order of right-hand sides

(iii) Create unique intervals for constraints that overlap

(iv) In S, replace original constraints by disjunction of constraints for modified constraints in overlapping intervals

Once the integer variables have been reduced to user-defined types and the constraints in the problem formulation S have been replaced with the appropriate combination of propositions, the full SAT instance can be created using the *at most*

one and the *at least one* clauses, in the same fashion as explained in Section 4.1. For a specifications where there is significant use of integer constraints, the use of Mixed Integer Programming (MIP) solvers could be better suited for completeness and consistency analysis. This option is investigated in Section 6.

4.3 Complete Translation Algorithm

The basic translation principles have been explained in the previous sections. The complete translation algorithm can now be given, for a single machine:

(i) Create problem instance S depending on the property to be checked (consistency or completeness), as explained in Section 3

(ii) Replace function machine calls with extra rules

(iii) Replace symbolic right-hand sides with values from the chosen configuration

(iv) Reduce integer variables to user-defined type variables, as explained in Section 4.2

(v) Iterate through all monitored variables and create *at least one* clauses and *at most one* clauses, as explained in section 4.1

(vi) Convert problem formulation S to conjunctive normal form and create the full SAT instance, as explained in Section 4.1

5 Example

The translation presented in this paper is implemented in the TASM toolset. The resulting SAT problem is automatically analyzed using the open source SAT4J SAT solver [11]. The SAT4J solver is a Java-based solver which can be integrated seamlessly into any Java application. The TASM toolset [17] provides the option to solve the completeness and consistency problems directly, without requiring the user to know that the specification is being translated to SAT. The toolset also provides the capability to "export" the generated SAT problem, so that the problem can be analyzed and solved outside of the toolset. The toolset was used to analyze the consistency and completeness of two examples, the production cell system [15] and an electronic throttle controller [13]. For these two examples, the performance of the translation algorithm and the feasibility of using a SAT solver proved adequate. As an example, the translation for a machine definition is given. The sample machine specification is extracted from production cell system case study. The machine definition is for the 'loader' component, which is the component of the system responsible for putting blocks on the feed belt. The machine specification, expressed in the TASM language, is shown in Listing 1.

For the verification of completeness, the translation to SAT, for initial conditions where $number = 5$, yielded 7 unique propositions:

Listing 1 Definition of the loader machine

```
R1: Empty Belt
{
  t     := 2;
  power := 200;

  if loaded_blocks < number - 1 and feed_belt = empty and feed_block = notavailable  then
    feed_belt       := loaded;
    loaded_blocks   := loaded_blocks + 1;
    loader_sensor!;
}

R2: Load Last Block
{
  t     := 2;
  power := 200;

  if loaded_blocks = number - 1 and feed_belt = empty and feed_block = notavailable  then
    feed_belt       := loaded;
    loaded_blocks   := loaded_blocks + 1;
    loader          := notavailable;
    loader_sensor!;
}

R3: Loaded Belt
{
  t := next;

  if feed_belt = loaded and loaded_blocks < number then
    skip;
}
```

$$
\begin{aligned}
b_1 &: loaded\_blocks \ <= \ 3 \\
b_2 &: loaded\_blocks \ = \ 4 \\
b_3 &: loaded\_blocks \ >= \ 5 \\
b_4 &: feed\_belt \ = \ empty \\
b_5 &: feed\_belt \ = \ loaded \\
b_6 &: feed\_block \ = \ available \\
b_7 &: feed\_block \ = \ notavailable
\end{aligned}
$$

Once the mapping between between TASM variable values and SAT boolean propositions has been established, the rule guards, G_i, can be expressed in terms of boolean propositions. The completeness problem, S, is then constructed according to the definition is Section 3.3:

$$
\begin{aligned}
G_1 &\equiv b_1 \ \wedge \ b_4 \ \wedge \ b_7 \\
G_2 &\equiv b_2 \ \wedge \ b_4 \ \wedge \ b_7 \\
G_3 &\equiv b_5 \ \wedge \ (b_1 \ \vee \ b_2) \\
S &\equiv \neg(G_1 \ \vee \ G_2 \ \vee \ G_3)
\end{aligned}
$$

The complete translation to SAT, in CNF, yielded 13 total propositions:

$$
\begin{aligned}
S\ in\ CNF &\begin{cases} (\neg b_7 \ \vee\ \neg b_4 \ \vee\ \neg b_1) & \wedge \\ (\neg b_7 \ \vee\ \neg b_4 \ \vee\ \neg b_2) & \wedge \\ (\neg b_1 \ \vee\ \neg b_5) & \wedge \\ (\neg b_2 \ \vee\ \neg b_5) & \wedge \end{cases} \\
At\ least\ one\ clauses &\begin{cases} (b_1 \ \vee\ b_2 \ \vee\ b_3) & \wedge \\ (b_4 \ \vee\ b_5) & \wedge \\ (b_6 \ \vee\ b_7) & \wedge \end{cases} \\
At\ most\ one\ clauses &\begin{cases} (\neg b_1 \ \vee\ \neg b_2 \ \vee\ \neg b_3) & \wedge \\ (b_1 \ \vee\ \neg b_2 \ \vee\ \neg b_3) & \wedge \\ (\neg b_1 \ \vee\ b_2 \ \vee\ \neg b_3) & \wedge \\ (\neg b_1 \ \vee\ \neg b_2 \ \vee\ b_3) & \wedge \\ (\neg b_4 \ \vee\ \neg b_5) & \wedge \\ (\neg b_6 \ \vee\ \neg b_7) & \end{cases}
\end{aligned}
$$

The SAT problem resulting from the translation is relatively small and running it through the SAT4J solver yields a solution in negligible time. For this machine, the rule set is not complete. The TASM toolset uses the SAT4J solver to generate the counterexamples in which no rule is enabled. An assignment to propositions that makes the problem satisfiable is ($b_2 = true$, $b_4 = true$, $b_6 = true$) and all other propositions are assigned $false$. In terms of the TASM specification, the counterexample which is generated is the set ($loaded\_blocks = 4$, $feed\_belt = empty$, $feed\_block = available$). To check the consistency of the rule set for the 'loader' machine, the same set of proposition was generated, but the set of clauses grew to 159. However, many of the clauses were redundant, due to the long form used for the conversion to CNF. Future work in tool development will improve the translation to CNF by removing redundant clauses. Nevertheless, the SAT problem was verified to be unsatisfiable in negligible time. In other words, the rules of machine 'loader' are consistent. The preliminary results from the translation algorithm indicate that the performance of the translation algorithm might overshadow the performance of the SAT solver.

6 Conclusion and Future Work

In this paper, a translation from Abstract State Machine (ASM) specifications to the boolean satisfiability problem is given. The translation is performed in the context of the Timed Abstract State Machine (TASM) language, but the translation is equally applicable to standard ASM specifications and ASM derivatives. The translation is used to investigate completeness and consistency of the specification, for a single abstract state machine. Completeness and consistency of specifications were identified as important properties of specifications. The ability to verify these prop-

erties automatically, using a widely available and optimized tool, a *SAT* solver, is provided. This approach contrasts previous attempts using other languages, which have used special purpose verification algorithms. Previous attempts have motivated the use of special purpose algorithms to remove the need to generate a global reachability graph, as would be done in approaches based on model checkers. The translation proposed in this work also removes the need to generate a global reachability graph by constraining the analysis to a single machine and by considering only the structural properties of the specification. The big open question in this work is whether the use of a *SAT* solver to verify consistency and completeness is feasible for archetypical real-time system specifications. The number of propositions can grow exponentially, depending on the nature of the specification. Preliminary results indicate that the translation algorithm could undergo further optimization since it appears to be a bottleneck, compared to the time spent in the *SAT* solver. The translation algorithm will be analyzed in detail for time complexity and will be optimized accordingly.

6.1 Future Work

The translation given in this work maps TASM specifications to boolean formulas. The use of boolean formulas makes negation and manipulation of rule guards straightforward. The translation will be used for model-based test case generation, using existing approaches [10] [18] and existing coverage criteria for ASM specifications [4]. For rule guards that contain multiple integer variables, the use of *SAT* solvers might not be the most optimal approach. Mixed Integer Programming (MIP) solvers such as the open source GNU Linear Programming Kit (GLPK) could provide a viable alternative. The translation to an MIP solver would not require the reduction of the integer variables as proposed in this work since MIP solvers can handle a mix of boolean variables and integer variables. However, using an MIP solver would require a reformulation of the problem because the input of such solvers requires a conjunction of constraints. Handling disjunction of constraints can be expressed, using modeling tricks such as the "big M" approach [19] and introducing extra binary variables to relax and enforce disjunction of constraints. The use MIP solvers would also enable the analysis of specifications involving the use of decimal datatypes. Other solvers could also be used, such as PROLOG-based solvers. While most of the solvers address problems known to be at least NP-Hard, it would be interesting to understand the average case performance for archetypical specifications. This could lead to beneficial analysis results, regardless of the nature of the problem. Before embarking on the use of different types of solvers, the feasibility of the translation to *SAT* must be assessed. This will be achieved by designing benchmarks using archetypical specifications. Once a good set of benchmarks have been derived for the *SAT*-based approach, the same set of benchmarks can be reused for MIP solvers and PROLOG-based solvers.

References

[1] Börger, E., *Why Use Evolving Algebras for Hardware and Software Engineering?*, in: *Proceedings of the 22nd Seminar on Current Trends in Theory and Practice of Informatics, SOFSEM '95*, LNCS **1012** (1995).

[2] Börger, E. and R. Stärk, "Abstract State Machines," Springer-Verlag, 2003.

[3] Castillo, G. D., *Towards Comprehensive Tool Support for Abstract State Machines: The ASM Workbench Tool Environment and Architecture*, in: *Applied Formal Methods – FM-Trends 98*, LNCS **1641** (1999), pp. 311–325.

[4] Gargantini, A. and E. Riccobene, *ASM-based Testing: Coverage Criteria and Automatic Test Sequence Generation*, Journal of Universal Computer Science **7** (2001).

[5] Grieskamp, W., Y. Gurevich, W. Schulte and M. Veanes, *Generating Finite State Machines From Abstract State Machines*, in: *Proceedings of the 2002 ACM SIGSOFT international symposium on Software Testing and Analysis*, 2002, pp. 112–122.

[6] Grieskamp, W., L. Nachmanson, N. Tillmann and M. Veanes, *Test Case Generation from AsmL Specifications - Tool Overview*, in: *Abstract State Machines – ASM 2003* (2003).

[7] Gurevich, Y., B. Rossman and W. Schulte, *Semantic Essence of AsmL*, Theoretical Computer Science **343** (2005), pp. 370–412.

[8] Heimdahl, M. P. E. and N. G. Leveson, *Completeness and Consistency in Hierarchical State-Based Requirements*, Software Engineering **22** (1996), pp. 363–377.

[9] Heitmeyer, C. L., R. D. Jeffords and B. G. Labaw, *Automated Consistency Checking of Requirements Specifications*, ACM Transactions on Software Engineering and Methodology (TOSEM) **5** (1996), pp. 231–261.

[10] Khurshid, S. and D. Marinov, *TestEra: A Novel Framework for Automated Testing of Java Programs*, in: *Proceedings of the 16th International Conference on Automated Software Engineering (ASE)*, 2001.

[11] Leberre, D., *SAT4J: A Satisfiability Library for Java*, presentation available from http://sat4j.objectweb.com.

[12] Ouimet, M., *The TASM Language Reference Manual, Version 1.1*, Available from http://esl.mit.edu/tasm (2006).

[13] Ouimet, M., G. Berteau and K. Lundqvist, *Modeling an Electronic Throttle Controller using the Timed Abstract State Machine Language and Toolset*, in: *Modeling in Software Engineering*, LNCS **4364**, 2007.

[14] Ouimet, M. and K. Lundqvist, *The Hi-Five Framework and the Timed Abstract State Machine Language*, in: *Proceedings of the 27th IEEE Real-Time Systems Symposium - Work in Progress Session*, 2006.

[15] Ouimet, M. and K. Lundqvist, *Modeling the Production Cell System in the TASM Language* (2007), technical Report ESL-TIK-000209, Embedded Systems Laboratory, Massachusetts Institute of Technology.

[16] Ouimet, M. and K. Lundqvist, *The Timed Abstract State Machine Language: An Executable Specification Language for Reactive Real-Time Systems*, in: *Proceedings of the 15th International Conference on Real-Time and Network Systems (RTNS '07)*, 2007.

[17] Ouimet, M. and K. Lundqvist, *The Timed Abstract State Machine Toolset: Specification, Simulation, and Verification of Real-Time Systems*, in: *Proceedings of the 19th International Conference on Computer-Aided Verification (CAV'07)*, 2007.

[18] Pretschner, A. and H. Lötzbeyer, *Model Based Testing with Constraint Logic Programming: First Results and Challenges*, in: *Proceedings of the 2nd ICSE International Workshop on Automated Program Analysis, Testing and Verification (WAPATV)*, 2001.

[19] Richards, A. and J. How, *Mixed-integer Programming for Control*, in: *Proceedings of the 2005 American Control Conference (ACC '05)*, 2005.

[20] Sipser, M., "Introduction to the Theory of Computation," Springer-Verlag, 2003.

[21] Winter, K., *Model Checking for Abstract State Machines*, Journal of Universal Computer Science **3** (1997), pp. 689–701.