AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications[*]

Paolo Arcaini[1] [**], Angelo Gargantini[2], and Elvinia Riccobene[1]

[1] Dip. di Tecnologie dell'Informazione, Università degli Studi di Milano, Italy
parcaini@gmail.com, elvinia.riccobene@dti.unimi.it
[2] Dip. di Ing. Gestionale e dell'Informazione, Università di Bergamo, Italy
angelo.gargantini@unibg.it

Abstract. This paper presents *AsmetaSMV*, a model checker for Abstract State Machines (ASMs). It has been developed with the aim of enriching the ASMETA (ASM mETAmodeling) toolset – a set of tools for ASMs – with the capabilities of the model checker NuSMV to verify properties of ASM models written in the AsmetaL language. We describe the general architecture of AsmetaSMV and the process of automatically mapping ASM models into NuSMV programs. As a proof of concepts, we report the results of using AsmetaSMV to verify temporal properties of various case studies of different characteristics and complexity.

Key words: Abstract State Machines, Model Checking, NuSMV, ASMETA

1 Introduction

To tackle the growing complexity of developing modern software systems that usually have embedded and distributed nature, and more and more involve safety critical aspects, formal methods have been affirmed as an efficient approach to ensure the quality and correctness of the design. Formal methods provide several advantages when involved in software system engineering. They allow producing unambiguous specifications about the features and behavior of a system; they allow catching and fixing design errors and inconsistencies early in the design process; they allow applying formal analyses methods (validation and verification) that assure correctness w.r.t. the system requirements and guarantee the required system properties.

The Abstract State Machines (ASMs) [7] are nowadays acknowledged as a formal method successfully employed as systems engineering method that guides the development of complex systems seamlessly from requirements capture to their implementation. To be used in an efficient manner during the system development process, ASMs should be endowed with tools supporting the major

[*] This work is supported in part by the PRIN project D-ASAP (Dependable Adaptable Software Architecture for Pervasive computing).

[**] The author was partially supported by the STMicroelectronics project on *Model-driven methodologies and techniques for the design and analyses of embedded systems*

software life cycle activities: editing, simulation, testing, verification, model exchanging, etc. It is also mandatory that these tools have to be strongly integrated in order to permit reusing information about models.

The goal of the ASMETA (ASM mETAmodeling) project [3] was to engineer a standard metamodel-based modeling language for the ASMs, and to build a general framework suitable for developing new ASM tools and integrate existing ones. Up to now, the ASMETA tool-set [14, 3, 12] allows creation, storage, interchange, Java representation, simulation, testing, scenario-based validation of ASM models.

In this work, we present a new component, *AsmetaSMV*, that enriches the ASMETA framework with the capabilities of the model checker NuSMV [2] to verify properties of ASM models.

As discussed in Sect. 5, it is relatively clear that a higher level specification formalism as that provided in terms of ASMs enables a more convenient modeling than that provided by the language of a model checker as NuSMV. On the other hand, it is undoubted that a lower-level formalism will lead to more efficient model checking. However, we believe that a developing environment where several tools can be used for different purposes on the base of the same specification model can be much more convenient than having different tools, even if more efficient, working on input models with their own different languages. On the base of our experience on some case studies (as the Mondex or the Flash Protocol, see Sect. 5), having a model checker integrated with a powerful simulator provides great advantages for model analyses, especially in order to perform model and property validation. Indeed, verification of properties should be applied when a designer has enough confidence that the specification and the properties themselves capture all the informal requirements. By simulation (interactive or scenario driven [8]), it is possible to ensure that the specification really reflects the intended system behavior. Otherwise there is the risk that proving properties becomes useless, for example in case of property *vacuously* true [16]. Moreover, a simulator can help to reproduce counter examples provided by a model checker, which are sometimes hermetic to understand.

The paper is organized as follows. Sect. 2 presents related results. In Sect. 3, we briefly introduce the ASMETA framework and the NuSMV model checker. Sect. 4 describes the general architecture of AsmetaSMV and the process of automatically mapping ASM models into NuSMV programs. In Sect. 5, we report the results of using AsmetaSMV to verify temporal properties of various case studies of different characteristics and complexity. Sect. 6 concludes the paper.

2 Related Work

There are several attempts to translate ASM specifications to the languages of different model checkers. For explicit state model checkers as Spin, we can cite [11] and [10]. In [11], the authors show how to obtain Promela programs from simple ASMs in order to use Spin for test generation. The approach was significantly improved in [10] where the authors reported their experience in using Spin for verifying properties of CoreAsm specifications.

Regarding NuSMV, which is a symbolic model checker, a preliminary work was done by Spielmann [19]. It represents ASMs by means of a logic for computational graphs that can be combined with a Computation Tree Logic (CTL) property and together they can be checked for validity. This approach severely limits the ASMs that can be model checked. That work was overtaken by the research of Winter and Del Castillo, which is very similar to ours. In [20], the author discusses the use of the model checker SMV (Symbolic Model Verifier) in combination with the specification method of ASMs. A scheme is introduced for transforming ASM models into the language of SMV from ASM workbench specifications. These schema are very similar to the scheme we present later in this paper. The approach was later improved in [9] and applied to a complex case study in [21]. A comparison with their work is presented in Sect. 5. Our approach is very similar to theirs, although the starting notation is different. Moreover, their tools (both the ASM workbench and the translator) are no longer maintained and we were unable to use them.

Other approaches to model checking ASMs include works which perform a quasi-native model checking without the need of a translation to a different notation. For example, [15] presents a model checking algorithm for AsmL specifications. The advantages is that the input language is very rich and expressive, but the price is that the model checking is very inefficient and unable to deal with complex specifications, and it is not able to perform all the optimizations available for a well established technique as that of Spin or NuSMV. A mixed approach is taken by [5], which presents a way for model checking ASMs without the need of translating ASM specifications into the modeling language of an existing model checker. Indeed, they combine the model checker [mc]square with the CoreASM simulator which is used to build the state space.

3 Background

3.1 ASMETA toolset

The ASMETA (ASM mETAmodeling) toolset [14, 3, 13] is a set of tools for the ASMs developed by exploiting the Model-driven development (MDD) approach.

We started by defining a metamodel, the *Abstract State Machine Metamodel* (AsmM), as abstract syntax description of a language for ASMs. From the AsmM, by exploiting the MDD approach and its facilities (derivative artifacts, APIs, transformation libraries, etc.), we obtained in a generative manner (i.e. semi-automatically) several artifacts (an interchange format, APIs, etc.) for the creation, storage, interchange, access and manipulation of ASM models [12]. The AsmM and the combination of these language artifacts have led to an instantiation of the Eclipse Modeling Framework (EMF) for the ASM application domain. The resulting ASMETA framework provides a global infrastructure for the interoperability of ASM tools (new and existing ones) [13].

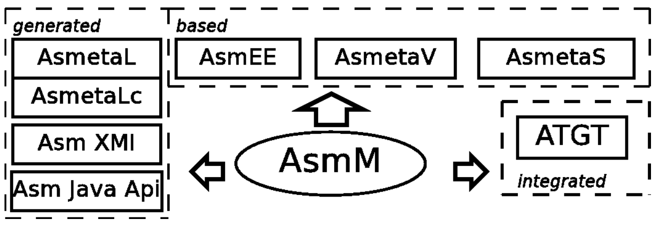

Fig. 1. The ASMETA tool set

The ASMETA tool set (see Fig. 1) includes (among other things) a textual concrete syntax, *AsmetaL*, to write ASM models (conforming to the AsmM) in a textual and human-comprehensible form; a text-to-model compiler, *AsmetaLc*, to parse AsmetaL models and check for their consistency w.r.t. the AsmM OCL constraints; a simulator, *AsmetaS*, to execute ASM models; the *Avalla* language for scenario-based validation of ASM models, with its supporting tool, the *AsmetaV* validator; the *ATGT* tool that is an ASM-based test case generator based upon the SPIN model checker; a graphical front-end called *ASMEE* (ASM Eclipse Environment) which acts as integrated development environment (IDE) and it is an Eclipse plug-in.

All the above artifacts/tools are classified in: *generated*, *based*, and *integrated*. Generated artifacts/tools are derivatives obtained (semi-)automatically by applying appropriate Ecore (i.e. the EMF metalanguage) projections to the technical spaces Javaware, XMLware, and grammarware. Based artifacts/tools are those developed exploiting the ASMETA environment and related derivatives; an example of such a tool is the simulator AsmetaS. Integrated artifacts/tools are external and existing tools that are connected to the ASMETA environment.

3.2 NuSMV

The NuSMV model checker [2], derived from the CMU SMV [17], allows for the representation of synchronous and asynchronous finite state systems, and for the analysis of specifications expressed in *Computation Tree Logic* (CTL) and *Linear Temporal Logic* (LTL), using Binary Decision Diagrams (BDD)-based and SAT-based model checking techniques. Heuristics are available for achieving efficiency and partially controlling the state explosion.

NuSMV is a transactional system in which the states are determined by the values of variables; transactions between the states are determined by the updates of the variables. A NuSMV model is made of three principal sections:

– VAR that contains variables declaration. A variable type can be *boolean*, *Integer* defined over intervals or sets, *enumeration* of symbolic constants.
– ASSIGN that contains the initialization (by the instruction *init*) and the update mechanism (by the instruction *next*) of variables. A variable can be not initialized and in this case, at the beginning NuSMV creates as many

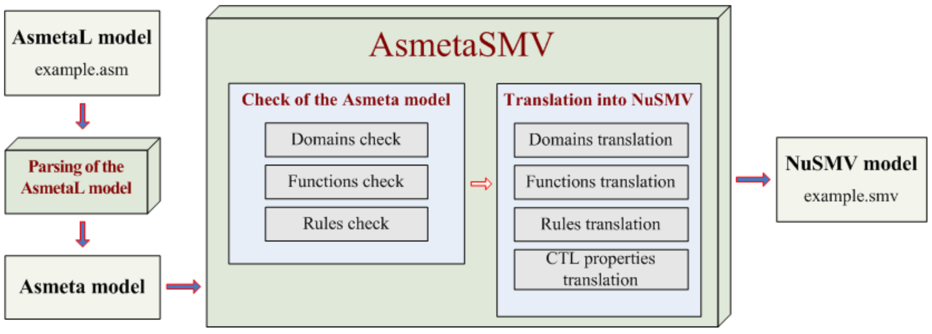

Fig. 2. Architecture of AsmetaSMV

states as the number of values of the variable type; in each state the variable assumes a different value. The next value can be determined in a straight way, or in a conditional way through the **case** expression.
- SPEC (resp. LTLSPEC) that contains the CTL (resp. LTL) properties to be verified.

In NuSMV it is possible to model *non deterministic behaviours* by (a) do not assigning any value to a variable that, in this case, can assume any value; (b) assigning to a variable a value randomly chosen from a set. It is also possible to specify *invariant conditions* by the command INVAR.

4 AsmetaSMV

AsmetaSMV has been developed as *based* tool of the ASMETA toolset, since it exploits some derivatives of the ASMETA environment. In particular, AsmetaSMV does not define its own input language, neither introduces a parser for a textual syntax. It reuses the parser defined for AsmetaL and reads the models as Java objects as defined by the ASMETA Java API. The aim of AsmetaSMV is that of enriching the ASMETA toolset with the capabilities of the model checker NuSMV. No knowledge of the NuSMV syntax is required to the user in order to use AsmetaSMV. To perform model checking over ASM models written in AsmetaL, a user must know, besides the AsmetaL language, only the syntax of the temporal operators.

Fig. 2 shows the general architecture of the tool. AsmetaSMV takes in input ASM models written in AsmetaL and checks if the input model is adequate to be mapped into NuSMV. Limitations are due to the model checker restriction over finite domains and data types. If this test fails, an exception is risen; otherwise, signature and transitions rules are translated as described in Sect. 4.1 and 4.2. The user can define temporal properties directly into the AsmetaL code as described in Sec. 4.3. We assume that the user provides the models in AsmetaL, but any other concrete syntax (like Asmeta XMI) could be used instead.

```
asm arity1_2
import ./StandardLibrary

signature:
  domain SubDom subsetof Integer
  enum domain EnumDom = {AA | BB}
  dynamic controlled foo1: Boolean -> EnumDom
  dynamic controlled foo2: Prod(SubDom ,
     EnumDom) -> SubDom
definitions:
  domain SubDom = {1..2}
```

```
MODULE main
  VAR
    foo1_FALSE: {AA,BB};
    foo1_TRUE: {AA,BB};
    foo2_1_AA: 1..2;
    foo2_1_BB: 1..2;
    foo2_2_AA: 1..2;
    foo2_2_BB: 1..2;
```

Fig. 3. AsmetaL model and NuSMV translation

4.1 Mapping of States

Domains. AsmetaL domains are mapped into their corresponding types in NuSMV. The only supported domains are: Boolean, Enum domains and Concrete domains whose type domains are Integer or Natural. Boolean and Enum domains are straightforwardly mapped into boolean and symbolic enum types of NuSMV. Concrete domains of Integer and Natural, instead, become integer enums in NuSMV, on the base of the concrete domain definitions.

Functions. For each AsmetaL dynamic *nullary* function (i.e. variable) a NuSMV variable is created. ASM *n-ary* functions must be decomposed into function locations; each location is mapped into a NuSMV variable. So, the cardinality of the domain of a function determines the number of the corresponding variables in NuSMV. The codomain of a function, instead, determines the type of the variable. Therefore, given an n-ary function *func* with domain $Prod(D_1, \dots, D_n)$, in NuSMV we introduce $\prod_{i=1}^{n} |D_i|$ variables with names $\mathsf{func\_elDom_1\_ \dots \_elDom}_n$, where $elDom_1 \in D_1, \dots, elDom_n \in D_n$.

Fig. 3 reports an example of two functions `foo1` of arity 1 and `foo2` of arity 2 and the result of the translation in NuSMV.

Controlled functions. They are updated by transitions rules. The initialization and the update of a dynamic location are mapped in the ASSIGN section through the *init* and *next* instructions. In Fig. 4, see the function `foo` as an example of a controlled function.

Monitored functions. Since their value is set by the environment, when mapped to NuSMV, monitored variables are declared but they are neither initialized nor updated. When NuSMV meets a monitored variable, it creates a state for each value of the variable. Values of monitored locations are set at the beginning of the transaction, that is before the execution of the transition rules; this means that transition rules deal with the monitored location values of the current state and not of the previous one. Therefore, when a monitored variable is used in the ASSIGN section (this means that, in AsmetaL, the corresponding monitored location occurs on the right side of a transition rule), its correct value is obtained through the *next* expression.

```
asm contrMon
import ./StandardLibrary
import ./CTLLibrary
signature:
  dynamic monitored mon: Boolean
  dynamic controlled foo: Boolean
definitions:
  //axiom for simulation
  axiom over foo: foo = mon
  //property to translate into NuSMV
  axiom over ctl: ag(foo = mon)

  main  rule r_Main = foo := mon
default init s0:
  function foo = mon
```

```
MODULE main
  VAR
    foo: boolean;
    mon: boolean;
  ASSIGN
    init(foo) := mon;
    next(foo) := next(mon);
SPEC  AG(foo = mon);
```

Fig. 4. Controlled and monitored functions

This is shown by an example reported in Fig. 4 where the axiom checks that the controlled function `foo` is always equal to the monitored function `mon`. The correct NuSMV translation reports a CTL property, equivalent to the axiom, which checks that the NuSMV model keeps the same behaviour of the AsmetaL model.

Static and derived functions. Their value is set in the *definitions* section of the AsmetaL model and never changes during the execution of the machine. AsmetaSMV does not distinguish between static and derived functions, that, in NuSMV, are expressed through the DEFINE statement, as shown in Fig. 5. To obtain a correct NuSMV code, static and derived functions must be fully specified (i.e. specified in all the states of the machine), otherwise, NuSMV signals that the conditions of these function definitions are not exhaustive.

```
asm staticDerived
import ./StandardLibrary
signature:
  domain MyDomain subsetof Integer
  dynamic monitored mon1: Boolean
  dynamic monitored mon2: Boolean
  static stat: MyDomain
  derived der: Boolean
definitions:
  domain MyDomain = {1..4}
  function stat = 2
  function der = mon1 and mon2
```

```
MODULE main
  VAR
    mon1: boolean;
    mon2: boolean;
  DEFINE
    stat:= 2;
    der:= (mon1 & mon2);
```

Fig. 5. Static/derived functions

4.2 Mapping of Transition Rules

ASMs and NuSMV differ in the way they compute the next state of a transition, and such difference is reflected in their syntaxes as well.

In ASM, at each state, every enabled rule is evaluated and the update set is built by collecting all the locations and next values to which locations must be updated. The same location can be assigned to different values in several points of the specifications and the typical syntax of a single guarded update is **if** *cond* **then** $var' := val$.

In NuSMV, at each step, for every variable, the next value is computed by considering *all* its possible guarded assignments. The form of a guarded update is var' := **case** *cond*1 **then** *val*1 **case** *cond*2 **then** *val*2... which lists all the possible next values for the location.

In order to translate from AsmetaL to NuSMV, our translation algorithm visits the ASM specification. It starts from the main rule and by executing a depth visit of all the rules it encounters, it builds a *conditional update map*, which maps every location to its update value together with its guard. A global stack *Conds* (initialized to *true*) is used to store the conditions of all the outer rules visited. For each rule constructor, a suitable visit procedure is defined.

Update rule The update rule syntax is $l := t$, where l is a location and t a term.

The visit algorithm builds c as the conjunction of all the conditions on the *Conds* stack and adds to the *conditional update map* the element $l \rightarrow (c, t)$

Conditional Rule The conditional rule syntax is:

$$\textbf{if} \quad cond \quad \textbf{then} \quad R_{then} \quad \textbf{else} \quad R_{else} \quad \textbf{endif}$$

where *cond* is a boolean condition and R_{then} and R_{else} are transition rules. If *cond* is true R_{then} is executed, otherwise R_{else} is executed.
The visit algorithm works as follows:

- *cond* is put on stack *Conds* and rule R_{then} is visited; in such a way all the updates contained in R_{then} are executed only if *cond* is true;
- *cond* is removed from stack *Conds*.
- If else branch is not null:
    - condition ¬*cond* is put on stack *Conds* and rule R_{else} is visited; in such a way all the updates contained in R_{else} are executed only if *cond* is false;
    - ¬*cond* is removed from stack *Conds*.

For example, the conditional update map of AsmetaL code shown in Fig. 6 is the following:

Location	**Condition**	**Value**
foo	mon ∧ ¬ mon2	AA
	mon ∧ mon2	BB
foo1	*true*	AA

```
asm condRule
import ./StandardLibrary

signature:
  enum domain EnumDom = {AA| BB| CC}
  dynamic monitored mon: Boolean
  dynamic monitored mon2: Boolean
  dynamic controlled foo: EnumDom
  dynamic controlled foo1: EnumDom

definitions:
  main  rule r_Main =
    par
      foo1 := AA
      if(mon) then
        if(mon2) then
          foo := BB
        else
          foo := AA
        endif
      endif
    endpar
```

```
MODULE main
  VAR
    foo: {AA, BB, CC};
    foo1: {AA, BB, CC};
    mon: boolean;
    mon2: boolean;
  ASSIGN
    next(foo) :=
      case
        next(mon) & !(next(mon2)): AA;
        next(mon) & next(mon2): BB;
        TRUE: foo;
      esac;
    next(foo1) := AA;
```

Fig. 6. Conditional Rule mapping

Choose rule The choose rule syntax is:

$$\begin{array}{l}\textbf{choose } v_1 \textbf{ in } D_1, \dots, v_n \textbf{ in } D_n \textbf{ with } G_{v_1,\dots,v_n} \textbf{ do}\\ \qquad R_{v_1,\dots,v_n}\\ \qquad [\textbf{ifnone } R_{ifnone}]\end{array}$$

where $v_1, \dots, v_n$ are logical variables and $D_1, \dots, D_n$ their domains. $G_{v_1,\dots,v_n}$ is a boolean condition over $v_1, \dots, v_n$. $R_{v_1,\dots,v_n}$ is a rule that contains occurrences of $v_1, \dots, v_n$. Optional branch **ifnone** contains the rule R_{ifnone} that must be executed if there are not values for variables $v_1, \dots, v_n$ that satisfy $G_{v_1,\dots,v_n}$
In NuSMV the logical variables become variables whose value is determined through an INVAR specification that reproduces the nondeterministic behavior of the choose rule. For each values tuple $d_1^{j_1}, \dots, d_n^{j_n}$ with $d_1^{j_1} \in D_1, \dots, d_n^{j_n} \in D_n$, the algorithm adds to the stack the condition $G_{d_1^{j_1},\dots,d_n^{j_n}}$ and visits rule $R_{d_1^{j_1},\dots,d_n^{j_n}}$[3].

Other rules In addition to the update, conditional and choose rules, the other rules that are supported by AsmetaSMV are: macrocall rule, block rule, case rule, let rule and forall rule. They are not reported here. Details can be found in [4].

[3] $G_{d_1^{j_1},\dots,d_n^{j_n}}$ and $R_{d_1^{j_1},\dots,d_n^{j_n}}$ are the condition and the rule where the variables $v_1, \dots, v_n$ have been replaced with the current values $d_1^{j_1}, \dots, d_n^{j_n}$.

4.3 Property Specification

AsmetaSMV allows the user to declare CTL/LTL properties directly in the axiom section of an AsmetaL model.

In AsmetaL, the syntax of an axiom is:

$$\textbf{axiom over}\; id_1, \ldots, id_n : \quad ax_{id_1,\ldots,id_n}$$

where $id_1, \ldots, id_n$ are names of domains, functions or rules; $ax_{id_1,\ldots,id_n}$ is a boolean expression containing occurrences of $id_1, \ldots, id_n$.

In NuSMV, CTL [resp. LTL] properties are declared through the keyword SPEC [resp. LTLSPEC]:

$$\textbf{SPEC} \quad p_{CTL} \qquad\qquad [\text{ resp. } \textbf{LTLSPEC} \quad p_{LTL}]$$

where p_{CTL} [resp. p_{LTL}] is a CTL [resp. LTL] formula.

The syntax of a CTL/LTL property in AsmetaL is:

$$\textbf{axiom over}\; [ctl \mid ltl] : \quad p$$

where the over section specifies if p is a CTL or a LTL formula.
In order to write CTL/LTL formulas in AsmetaL, we have created the libraries *CTLlibrary.asm* and *LTLlibrary.asm* where, for each CTL/LTL operator, an equivalent function is declared. The following table shows, as example, all the CTL functions.

NuSMV CTL operator	**AsmetaL CTL function**
EG p	static eg: Boolean → Boolean
EX p	static ex: Boolean → Boolean
EF p	static ef: Boolean → Boolean
AG p	static ag: Boolean → Boolean
AX p	static ax: Boolean → Boolean
AF p	static af: Boolean → Boolean
E[p U q]	static e: Prod(Boolean, Boolean) → Boolean
A[p U q]	static a: Prod(Boolean, Boolean) → Boolean

AsmetaL code in Fig. 7 contains three CTL properties and their translation into NuSMV.

4.4 Property Verification

AsmetaSMV allows model checking an AsmetaL specification by translating it to the NuSMV language and directly run the NuSMV tool on this translation to verify the properties. The output produced by NuSMV is pretty-printed, replacing the NuSMV variables with the corresponding AsmetaL locations: it is our desire, in fact, to hide as much as possible the NuSMV syntax to the user.

The output produced by model checking the AsmetaL model shown in Fig.7 is reported below. The first two properties are proved true: location `foo(AA)`,

```
asm ctlExample
import ./StandardLibrary
import ./CTLlibrary

signature:
   enum domain EnumDom = {AA | BB}
   controlled foo: EnumDom -> Boolean
   monitored mon: Boolean

definitions:
   //true
   axiom over ctl: ag(foo(AA) iff
                    ax(not(foo(AA))))
   //true
   axiom over ctl: ag(not(foo(AA)) iff
                         ax(foo(AA)))
   //false. Gives counterexample.
   axiom over ctl: not(ef(foo(AA) !=
                                 foo(BB)))

   main rule r_Main =
      par
         foo(AA) := not(foo(AA))
         if(mon) then
            foo(BB) := not(foo(BB))
         endif
      endpar

default init s0:
   function foo($x in EnumDom) = true
```

```
MODULE main
   VAR
      foo_AA: boolean;
      foo_BB: boolean;
      mon: boolean;
   ASSIGN
      init(foo_AA) := TRUE;
      init(foo_BB) := TRUE;
      next(foo_AA) := !(foo_AA);
      next(foo_BB) :=
         case
            next(mon): !(foo_BB);
            TRUE: foo_BB;
         esac;
SPEC  AG(foo_AA <-> AX(!(foo_AA)));
SPEC  AG(!(foo_AA) <-> AX(foo_AA));
SPEC  !(EF(foo_AA != foo_BB));
```

Fig. 7. CTL property translation

in fact, changes its value at each step. The third property, instead, is proved false as shown by the counterexample: a state exists where locations `foo(AA)` and `foo(BB)` are different (State: 1.2). Note that the pretty printer substitutes foo_AA with `foo(AA)`.

```
> AsmetaSMV ctlExample.asm
Checking the AsmetaL spec: OK
Translating to ctlExample.smv: OK
Executing NuSMV -dynamic -coi ctlExample.smv
*** This is NuSMV 2.4.0 (compiled on Sat Oct  4 10:17:49 UTC 2008)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

-- specification AG (foo(AA) <-> AX !foo(AA))  is true
-- specification AG (!foo(AA) <-> AX foo(AA))  is true
-- specification !(EF foo(AA) != foo(BB))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  foo(AA) = 1
  foo(BB) = 1
  mon = 0
-> Input: 1.2 <-
-> State: 1.2 <-
  foo(AA) = 0
```

5 Case studies

AsmetaSMV has been tested on five case studies; the complete description of our tests can be found in [4] and are available at [3]. The first two case studies we have analyzed are two problems described in [6]:

1. A system made of two traffic lights placed at the beginning and at the end of an alternated one-way street; both traffic lights are controlled by a computer.
2. An irrigation system composed by a small sluice, with a rising and a falling gate, and a computer that controls the sluice gate.

For both problems we have written *ground* and *refined* model; in each model we have declared safety and liveness properties to test the correctness of the model.

Another case study we have analyzed is the Mondex protocol ([1]). The Mondex protocol implements electronic cash transfer between two purses; the transfer of money is implemented through the sending of messages over a lossy medium that can be, for example, a device with two slots or an Internet connection. We have written the AsmetaL model for one of the refinement steps described in [18]; a liveness property has helped us to discover that the model can enter in a deadlock state. Two different solutions are proposed in [4].

We have also analyzed the taxi booking problem: in a city some clients can request one or more taxis to a central that must satisfy all the requests. The taxis must bring the clients where they want to go. For this problem we had previously developed a NuSMV model (let's call it *originalNuSMV*); now we have developed an AsmetaL model containing the same properties that we wrote in the *originalNuSMV*. We have been able to compare the *originalNuSMV* code with the code obtained from the translation of the AsmetaL model (let us call it *mappedNuSMV*). We have seen that, for the same problem, it is easier to write an AsmetaL code rather than a NuSMV one: the ASMs in fact, thanks to a wide set of transition rules, are much more expressive than NuSMV. The verification of the properties in *originalNuSMV* and in *mappedNuSMV* gave the same results. Obviously this cannot be considered as a demonstration of the correctness of the mapping, but shows that, for a problem, there are different equivalent models. Generally, the code obtained from a mapping is more computational onerous than a code written directly in NuSMV; the mapping, in fact, introduces some elements that can be avoided in the direct encoding.

Finally, we have applied our tool to the flash cache coherence protocol, which integrates support for cache coherent shared memory for a large number of interconnected processing nodes. Starting from the specifications published by Winter [21] and by Farahbod at alt. [10], we have written the AsmetaL specification for the protocol together with its safety properties. This model is available in the

ASMETA repository. By means of the ASMETA simulator and the validator we were able to correct some defects in our specifications even before trying to prove the properties. A problem of vacuity detection also has been arisen. At the end, we were able to prove the three properties in less than 1 second for both the protocol versions with 2 nodes and 1 and 2 lines. A detailed comparison with [21] and [10] is however difficult since we were unable to run neither Asm2SMV which is no longer maintained, nor the Coreasm to Spin plug-in which is not published yet. With respect to [21], our running times are much lower, but we ran the experiments on a faster machine. In terms of the BDD size of the resulting NuSMV model, we found that the specification with 1 line has similar size while our specification with 2 lines had a much smaller BDD size. Our experiments were much faster than that in [10], too, but only for one property we can actually compare our results with theirs, since they used the model checker Spin mainly to find faults in the original specification but we were unable to reproduce the same faults.

6 Conclusions

This work is part of our ongoing effort in developing a set of tools around ASMs for model validation and verification. We here describe how the ASMETA toolset has been enriched with model checking facilities to verify temporal properties of ASM models encoded in the AsmetaL language. By means of case studies of different complexity, we provide evidence of the importance of having simulation and model checking capabilities integrated within a unique environment. Indeed, the combined use of both tools can facilitate the verification process, since it may be sometimes useful to discover which system behavior is hidden behind a property to verify in order to better formulate it and easily prove it. As future plan, we intend to improve AsmetaSMV to handle turbo ASMs. Moreover, we plan to extend AsmetaSMV in order to allow the translation of counter examples produced by NuSMV to Avalla [8], the language we use to perform scenario driven validation of ASMs. The counter examples would constitute a set of wrong scenarios representing incorrect behaviors of the system, and they could be replayed later to check that corrected models do not exhibit those incorrect behaviors.

Acknowledgments R. Farahbod sent us the CoreAsm specification of the flash coherence protocol, while K. Winter helped us in trying to make Asm2SMV [21] work on our computers.

References

1. Mastercard international inc.: Mondex. `http://www.mondex.com/`.
2. The NuSMV website. `http://nusmv.itc.it/`.
3. The ASMETA website. `http://asmeta.sourceforge.net/`, 2006.

4. P. Arcaini, A. Gargantini, and E. Riccobene. AsmetaSMV: a model checker for AsmetaL models. tutorial. TR 120, DTI Dept., Univ. of Milan, 2009.
5. J. Beckers, D. Klünder, S. Kowalewski, and B. Schlich. Direct support for model checking abstract state machines by utilizing simulation. In E. Börger, M. J. Butler, J. P. Bowen, and P. Boca, editors, *ABZ '08*, volume 5238 of *LNCS*, pages 112–124. Springer, 2008.
6. E. Börger. The Abstract State Machines Method for High-Level System Design and Analysis. Technical report, BCS Facs Seminar Series Book, 2007.
7. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.
8. A. Carioni, A. Gargantini, E. Riccobene, and P. Scandurra. A scenario-based validation language for ASMs. In E. Börger, M. Butler, J. P. Bowen, and P. Boca, editors, *ABZ '08*, volume 5238 of *LNCS*, pages 71–84. Springer, 2008.
9. G. D. Castillo and K. Winter. Model checking support for the ASM high-level language. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000*, volume 1785 of *LNCS*, pages 331–346. Springer, 2000.
10. R. Farahbod, U. Glässer, and G. Ma. Model checking coreasm specifications. In A. Prinz, editor, *Proceedings of the ASM'07, The 14th International ASM Workshop*, 2007.
11. A. Gargantini, E. Riccobene, and S. Rinzivillo. Using Spin to generate tests from ASM specifications. In *Abstract State Machines, Advances in Theory and Practice*, number 2589 in LNCS, pages 263–277. Springer, 2003.
12. A. Gargantini, E. Riccobene, and P. Scandurra. A metamodel-based language and a simulation engine for abstract state machines. *J. UCS*, 14(12):1949–1983, 2008.
13. A. Gargantini, E. Riccobene, and P. Scandurra. Model-driven language engineering: The ASMETA case study. In *International Conference on Software Engineering Advances, ICSEA*, pages 373–378, 2008.
14. A. Gargantini, E. Riccobene, and P. Scandurra. Ten reasons to metamodel ASMs. In Jean-Raymond and U. Glässer, editors, *Rigorous Methods for Software Construction and Analysis*, number 5115 in LNCS. Springer, 2009.
15. M. Kardos. An approach to model checking asml specifications. In *Abstract State Machines*, pages 289–304, 2005.
16. O. Kupferman and M. Vardi. Vacuity detection in temporal model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(2):224–233, 2003.
17. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.
18. G. Schellhorn and R. Banach. A concept-driven construction of the mondex protocol using three refinements. In E. Börger, M. Butler, J. P. Bowen, and P. Boca, editors, *ABZ '08*, volume 5238, pages 57–70, Berlin, 2008. Springer.
19. M. Spielmann. Automatic verification of abstract state machines. In N. Halbwachs and D. Peled, editors, *CAV*, volume 1633 of *Lecture Notes in Computer Science*, pages 431–442. Springer, 1999.
20. K. Winter. Model Checking for Abstract State Machines. *Journal of Universal Computer Science (J.UCS)*, 3(5):689–701, 1997.
21. K. Winter. Towards a methodology for model checking ASM: Lessons learned from the FLASH case study. In Y. Gurevich, P. W. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines*, volume 1912 of *Lecture Notes in Computer Science*, pages 341–360. Springer, 2000.