Model Checking Abstract State Machines with Answer Set Programming

Calvin Kai Fan Tang and Eugenia Ternovska

School of Computing Science, Simon Fraser University, Burnaby, BC, Canada
{ctang, ter}@cs.sfu.ca

Abstract. The quality of a computer system can be enhanced by modelling its design before implementation and verifying its correctness. Abstract State Machines (ASMs) provide a mathematical framework for system modelling, while Model Checking is a technology for verification of system properties. Together, they form a powerful tool for checking systems. Bounded Model Checking (BMC) based on Answer Set Programming (ASP) is regarded as a competitive model checking approach due to its compact encodings of BMC problems. In this article we present BMC of ASM models based on ASP. We show how to translate an ASM and a temporal property into a logic program and solve the BMC problem for the ASM by computing an answer set for the corresponding logic program. Experimental results indicate that our method can outperform the existing SAT-based BMC of ASMs.

1 Introduction

Abstract State Machines (ASMs) [2] are a formal specification tool for modelling transition systems and are used in both industrial and academic settings and in various application domains. They enable designers to precisely capture the natural intuition of system behaviour from its informal requirements and to eliminate ambiguities, inconsistencies and incompleteness. ASM refinement techniques facilitate modelling on different levels of abstraction. In addition, different verification and validation techniques can be applied to ASMs because of their mathematical nature. Verification and validation are important in system development as they focus on ensuring system quality.

Model Checking is a highly effective approach to formal verification of finite transition systems. It is mainly used for detecting errors that are difficult to uncover through traditional simulation. Case studies, such as those in [4] and [17], demonstrate the feasibility of model checking of large system specifications. The first widely used method of model checking is *symbolic model checking* [13]. In this method, sets of states and transitions are represented as boolean functions using Ordered Binary Decision Diagrams (BDDs). However, some operations cannot be represented compactly as BDDs, and the size of the BDD representation of a boolean function is sensitive to the variable ordering. In response to those weaknesses, methods based on propositional satisfiability (SAT) sprang up due to the ability of SAT procedures to handle the space problem of BDD-based approaches. In particular, SAT-based *Bounded Model Checking* (BMC) [1] is considered as a complementary technique to BDD-based model checking. The idea of BMC is to search for a *counterexample* for a property whose length is bounded by a non-negative integer k. A BMC problem can be translated into a propositional satisfiability problem and solved with SAT checking techniques.

The SAT encoding of BMC was polynomial in the size of the given model, property and time bound[1]. The motivation to achieve a more succinct encoding brought the development of BMC using *Answer Set Programming* (ASP) (see e.g. [12]). ASP is a declarative logic programming paradigm for solving combinatorial search problems. In ASP, a problem is represented as a logic program whose answer sets stand for the solutions to the problem. A solution is determined by computing an answer set for the logic program using an answer set solver such as `SMODELS` [15]. The idea of ASP-based BMC is to describe models and properties in logic programs and reduce BMC problems to computing answer sets for logic programs. This approach leads to linear encodings of BMC problems, and in [9], it was shown to be competitive to SAT-based BMC with the well-known model checker NuSMV [5]. ASP-based BMC is very new and has potential for improvement, and therefore we believe that it is worth further exploration.

In this article we present a method of applying answer set programming to the bounded model checking of abstract state machines. We introduce a compact encoding of an ASM in a logic program and a way to solve a BMC problem for the ASM with ASP. In addition, we compare our method to SAT-based BMC of ASMs with NuSMV. Our main contribution lies in combining the notions of ASMs, BMC and ASP to develop a novel formal verification approach.

The rest of this article is organized as follows. In Section 2 we give an introduction to bounded model checking as well as temporal logic. In Section 3 we provide the necessary mathematical background for answer set programming. In Section 4 we explain in detail ASP-based BMC of ASMs. Section 5 reports our experimental results. We discuss related work in Section 6 and finish with our conclusions in Section 7.

2 Bounded Model Checking

For the introduction to bounded model checking, we will largely follow the description in [1]. In model checking, a model is described as a *Kripke structure*. A Kripke structure M is a quadruple (S,I,T,L), where S is the set of states, $I \subseteq S$ is the set of initial states, $T \subseteq S \times S$ is the transition relation, and L: $S \rightarrow P(A)$ is the labelling function, where A is the set of atomic propositions and $P(A)$ is the powerset over A. For each state $s \in S$, $L(s)$ is the set of atomic propositions which are true in s.

The behaviour of M is defined by *paths*. A path π of M is a state sequence $(s_0,s_1,...)$, where $s_i \in S$ and $T(s_i,s_{i+1})$ holds for all $i \geq 0$. The symbol $\pi(i)$ denotes

[1] A linear encoding in the size of the bound has recently been proposed in [10].

the i-th state of the path (i.e., s_i) and π_i denotes the suffix of π which starts at s_i. If $\pi(0)\in I$, π is called an *initialized* path.

In model checking, properties are specified in *temporal logic*, which is an extension of classical logic. We focus on *Linear Temporal Logic* (LTL). The syntax of LTL formulas is given as follows:

$$\psi ::= p\in A \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \mathbf{G}\psi \mid \mathbf{F}\psi \mid \mathbf{X}\psi.$$

We omit the binary temporal operators like the until operator **U** and the release operator **R**. We refer our readers to [6] for details. The semantics of LTL formulas is given in terms of paths of a Kripke structure. The relation that a path π satisfies a formula ψ ($\pi \models \psi$) is defined below:

$\pi \models p$ iff $p\in L(\pi(0))$ $\quad$ $\pi \models \neg\psi$ iff $\pi \not\models \psi$
$\pi \models \psi_1 \wedge \psi_2$ iff $\pi \models \psi_1$ and $\pi \models \psi_2$ $\quad$ $\pi \models \psi_1 \vee \psi_2$ iff $\pi \models \psi_1$ or $\pi \models \psi_2$
$\pi \models \mathbf{G}\psi$ iff $\pi_i \models \psi$ for all $i \geq 0$ $\quad$ $\pi \models \mathbf{F}\psi$ iff $\pi_i \models \psi$ for some $i \geq 0$
$\pi \models \mathbf{X}\psi$ iff $\pi_1 \models \psi$

A Kripke structure M satisfies a formula ψ ($M \models \psi$) iff all initialized paths of M satisfy ψ.

As mentioned above, the idea of bounded model checking is to search for a counterexample of length up to some non-negative integer k. A BMC problem is described by three components: a Kripke structure M, a temporal formula ψ (in the context of this article, an LTL formula ψ), and a time bound k. In BMC for LTL, finding a counterexample for ψ is equivalent to finding a finite prefix of an initialized path in M that satisfies $\neg\psi$. The formula $\neg\psi$ must be converted to *negation normal form*, in which negations only appear in front of atomic propositions. An LTL formula can be converted to negation normal form with these equivalences: $\neg(\psi_1 \wedge \psi_2) \equiv \neg\psi_1 \vee \neg\psi_2$, $\neg(\psi_1 \vee \psi_2) \equiv \neg\psi_1 \wedge \neg\psi_2$, $\neg\neg\psi \equiv \psi$, $\neg\mathbf{G}\psi \equiv \mathbf{F}\neg\psi$, $\neg\mathbf{F}\psi \equiv \mathbf{G}\neg\psi$ and $\neg\mathbf{X}\psi \equiv \mathbf{X}\neg\psi$.

A prefix of an initialized path that satisfies $\neg\psi$ is a counterexample for the formula ψ. The length of the prefix is restricted by the bound k. In practice, we start with a small value for k and gradually increase k until a counterexample is found, a pre-determined upper bound for k is reached, or the problem becomes intractable.

A finite prefix of a path may still correspond to an infinite path in a Kripke structure if it contains a transition looping from one state to some previous state or itself (see Fig. 1 for an example). In some cases, a loop is necessary for a prefix to be a counterexample for an LTL formula. For example, for the formula $\mathbf{F}p$, even if p does not hold in any state from s_0 to s_k in a prefix, without a loop, it is unknown whether p actually holds in any state beyond s_k, and thus the prefix is not sufficient to be a counterexample for $\mathbf{F}p$.

3 Answer Set Programming

Most of the discussion on answer set programming is extracted from [12]. Given a set of propositional *atoms* A, the syntax of *formulas* is defined as follows:

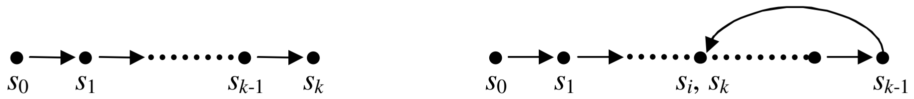

Fig. 1. Two possibilities for a prefix, prefix with no loop (left) and with a loop (right).

$$F ::= \top \mid \bot \mid p{\in}A \mid \mathit{not}\ F \mid F_1{,}F_2 \mid F_1{;}F_2.$$

A comma and a semicolon denote conjunction and disjunction, respectively. A *literal* is a formula that is an atom or its negation. A *logic program* Π consists of a finite set of *logic rules* of the form

$$F_1 \leftarrow F_2. \tag{1}$$

The formulas F_1 and F_2 are called the *head* and the *body* of the rule. If F_2 is $\top$, the rule can be abbreviated as F_1.

We define when a set of atoms X satisfies a formula F ($X \models F$) as follows:

$X \models \top$	$X \not\models \bot$
$X \models p$ iff $p{\in}X$	$X \models \mathit{not}\ F$ iff $X \not\models F$
$X \models F_1{,}F_2$ iff $X \models F_1$ and $X \models F_2$	$X \models F_1{;}F_2$ iff $X \models F_1$ or $X \models F_2$

X is said to satisfy a rule of the form (1) ($X \models F_1 \leftarrow F_2$) iff $X \models F_1$ whenever $X \models F_2$. X satisfies a program Π ($X \models \Pi$) iff it satisfies every rule in Π.

The *reduct* of a formula F in relation to a set of atoms X, denoted by F^X, is defined as follows:

$F^X = F$ if F is an atom, $\top$ or $\bot$	$(\mathit{not}\ F)^X = \bot$ if $X \models F$ and $\top$ otherwise
$(F_1{,}F_2)^X = {F_1}^X{,}{F_2}^X$	$(F_1{;}F_2)^X = {F_1}^X{;}{F_2}^X$

The *reduct* of a program Π in relation to X, denoted by Π^X, is the set of rules ${F_1}^X \leftarrow {F_2}^X$ for all rules $F_1 \leftarrow F_2$ in Π. X is called an *answer set* for a program Π iff X satisfies Π^X and is minimal among those that satisfy Π^X.

A *constraint* is a rule of the form

$$\bot \leftarrow F. \tag{2}$$

The head $\bot$ is optional. A set of atoms X satisfies a rule of the form (2) iff X does not satisfy F. A *cardinality expression* has the form

$$l \leq \{F_1, ..., F_n\} \leq u. \tag{3}$$

The symbols l and u are integers called the lower and upper bound, respectively. Cardinality expressions are useful for expressing choices in programs. Intuitively, a set of atoms X satisfies a cardinality expression of the form (3) iff the number of F_i satisfied by X is between l and u inclusive.

We can compute answer sets for a logic program using an answer set solver, such as `SMODELS`. `SMODELS` requires the tool `LPARSE` [16] as a preprocessor of programs. In `LPARSE`, `:-` is used for $\leftarrow$, and every rule ends with a period.

Consider the following program:

$$p_1 \leftarrow \mathit{not}\ p_2 \qquad p_2 \leftarrow \mathit{not}\ p_3 \qquad p_3 \leftarrow \mathit{not}\ p_4. \tag{4}$$

A group of rules which exhibit a pattern can be compactly described with variables. Variables must be in capital case. The program (4) can be represented in `LPARSE` by the lines

```
index(1..3).
p(I) :- not p(I+1), index(I).
```

The symbol `index` is called a *domain predicate*, which restricts the range of values a variable can have. The range `1..3` defines the extent of `index` and tells `LPARSE` which values should be used to replace the occurrences of the variable `I`. The process of substituting values for variables is called *grounding*, which is the primary task `LPARSE` performs.

4 Mapping BMC to ASP

We detail our method of applying answer set programming to bounded LTL model checking of abstract state machines. Assume that we are given an ASM model M, an LTL formula ψ to be verified, and a time bound k. We encode the behaviour of M up to k state transitions as a logic program $\Pi(M,k)$ and the negation of ψ as a program $\Pi(\neg\psi,k)$. Then we search for a counterexample in M for ψ by computing an answer set for the program $\Pi(M,k) \cup \Pi(\neg\psi,k)$.

Before discussing the construction of the programs, we introduce the two domain predicates *valid_state* and *has_next_state*, which will be used extensively in the rest of this section. The former is the domain for the states that are up to k transitions away from an initial state in a prefix, while the latter is the domain for the states that have a successor state in a prefix. They are defined as follows:

$$\mathit{valid\_state}(0..k)$$
$$\mathit{has\_next\_state}(0..k\text{-}1) \leftarrow k > 0.$$

4.1 Constructing Program $\Pi(M,k)$

An ASM has a *signature* Σ, which consists of a set of *sorts* and a set of *functions* [18]. Each function has an *arity* n, i.e., f: $D_1 \times ... \times D_n \rightarrow D_p$, where D_j ($1 \le j \le n$) and D_p are sorts. The signature Σ also contains the constants *true*, *false* and *undef*, the equality operator $=$, the sort *Boolean* and the operators $\wedge$, $\vee$ and $\neg$. A state s of Σ is a collection of *universes* D_j^s for each sort D_j in Σ and the interpretations of the functions in Σ over the universes.

Let the signature of M be Σ_M. We assume that each sort in Σ_M is associated with a fixed universe before model checking is applied to M. Dynamic extension of universes is not allowed, i.e., elements that are not contained in any of the universes cannot be introduced during the execution of M. In addition, we do not address the case in which M contains inconsistent updates, i.e., updates that associate different values with the same location in the same update set. It would be interesting to study the possibility of extending our method to handle extensible universes and inconsistent updates in the future.

We divide the task of constructing the program $\Pi(M,k)$ into four parts: sorts, functions, terms and transition rules.

Sorts. Each sort D_p in Σ_M is associated with a universe U_p. Let $\{c_1, ..., c_l\}$ be a set of constants whose interpretations are pairwise different elements in U_p. For each c_j, where $1 \leq j \leq l$, add the logic rule

$$d_p(c_j).$$

The symbol d_p is the domain predicate for sort D_p.

Functions. For each function f: $D_1 \times ... \times D_n \rightarrow D_p$ in Σ_M, f is either a *static* function which has a fixed interpretation, a *controlled* function whose interpretation can only be changed by the transition rules of M, or a *monitored* function whose interpretation can only be changed by the external environment. We differentiate the three cases as follows, where $Val_{s,\zeta}(t)$ denotes the value of the term t in state s with *variable assignment* ζ:

- If f is static, and $Val_{s,\zeta}(f(c_1,...,c_n)) = Val_{s,\zeta}(c_p)$ for all s and ζ, where each c_j is a constant of D_j for j=1,...,n and p, add the logic rule
$$f(c_1,...,c_n,c_p).$$
- If f is controlled, and $Val_{s_0,\zeta}(f(c_1,...,c_n)) = Val_{s_0,\zeta}(c_p)$ in every initial state s_0 of M, add the logic rule
$$f(c_1,...,c_n,c_p,0).$$
If no fixed value is given to $f(c_1,...,c_n)$ for all initial states, add the following logic rule which randomly picks one (and only one) constant of D_p to be equal to $f(c_1,...,c_n)$ in an initial state of M:
$$1 \leq \{f(c_1, ..., c_n, c_{p1}, 0), ..., f(c_1, ..., c_n, c_{pl}, 0)\} \leq 1,$$
where c_{pj} is a constant of D_p for $1 \leq j \leq l$.
If the value of a location of f is not updated in a state, it remains unchanged for the next state. Let the atom $f\_upd(c_1,...,c_n,i)$ be in an answer set iff the value of the location $(f,(Val_{s_i,\zeta}(c_1),..., Val_{s_i,\zeta}(c_n)))$ is updated in state s_i, where $0 \leq i \leq k$-1. Add the logic rule
$$\begin{aligned} f(X_1,...,X_n,X_p,I{+}1) \leftarrow\ & not\ f\_upd(X_1,...,X_n,I),\ f(X_1,...,X_n,X_p,I),\\ & d_1(X_1),\ ...,\ d_n(X_n),\ d_p(X_p),\ has\_next\_state(I). \end{aligned}$$
The symbols d_1,...,d_n and d_p are domain predicates for D_1,...,D_n and D_p, respectively.

- If f is monitored, the initial value of $f(c_1,...,c_n)$ is handled in the same way as a controlled function. A monitored function changes non-deterministically. To encode this behaviour, add the logic rule
$$\begin{aligned} 1 \leq \{\ & f(X_1,...,X_n,c_{p1},I{+}1),...,f(X_1,...,X_n,c_{pl},I{+}1)\ \} \leq 1 \leftarrow \\ & d_1(X_1),\ ...,\ d_n(X_n),\ has\_next\_state(I). \end{aligned}$$

Terms. *Terms* are syntactic expressions defined as follows:

- A variable is a term.
- If f: $D_1 \times ... \times D_n \rightarrow D_p$ is a function, and $t_1,...,t_n$ are terms of sorts $D_1,...,D_n$, respectively, then $f(t_1,...,t_n)$ is a term of D_p.
- If v is a variable, and $g(v)$ and $s(v)$ are terms of the sort *Boolean*, then $(\exists v{:}g(v))\ s(v)$ and $(\forall v{:}g(v))\ s(v)$ are terms, which are called *first-order terms*, with *head variable* v, *guard* $g(v)$ and *body* $s(v)$.

Let the transformation of terms be a function E_t, which takes a term t and an integer i identifying a state as parameters.

- A variable v of sort D_p is transformed into a logic program variable:

  $$E_t(v,i) = X \text{ for all } i,$$

  where X acts as a placeholder for the value of v. The transformation also produces the atom $d_p(X)$, where d_p is the domain predicate for D_p. We refer to the atoms introduced during the transformation of a term as 'side atoms'. If a logic program variable which represents the value of a term, such as X, appears in a logic rule, the side atoms for the term must be present in the body of the rule. We use the notation <*side atoms*> to denote the list of all side atoms created during the transformation of the terms in a logic rule.
- For a function application $f(t_1,...,t_n)$, if f is a static nullary function, i.e. a constant c, then $f(t_1,...,t_n)$ is c and is not affected by the transformation:

  $$E_t(c,i) = c \text{ for all } i.$$

  If f: $D_1 \times ... \times D_n \rightarrow D_p$ is not a constant, $f(t_1,...,t_n)$ is transformed into a logic program variable:

  $$E_t(f(t_1,...,t_n),i) = X \text{ for all } i,$$

  where X is a placeholder for the value of $f(t_1,...,t_n)$. If f is static, the side atoms are $f(E_t(t_1,i),...,E_t(t_n,i),X)$ and $d_p(X)$. If f is controlled or monitored, the side atoms are $f(E_t(t_1,i),...,E_t(t_n,i),X,i)$ and $d_p(X)$.
- A function application in which the function is supported by the answer set tool in use is handled differently. For example, all answer set tools support the operators $=$, $\wedge$, $\vee$ and $\neg$. These operations should be transformed into the appropriate syntax of the answer set tool. For instance, the equality $f(c_1){=}c_2$, where f: $D_1 \rightarrow D_2$ is a controlled unary function and c_1 and c_2 are constants of sorts D_1 and D_2 respectively, is translated into the expression $X == c_2$ in `LPARSE` with side atoms $f(c_1,X,i)$ and $d_2(X)$.
- A first-order term $(\exists v{:}g(v))\ s(v)$ is transformed as follows. We say that a first-order term t_f 'encloses' another one $t_{f'}$ if $t_{f'}$ appears in the guard or the body of t_f. Let $v_1,...,v_n$ be the head variables of the first-order terms which enclose the term $(\exists v{:}g(v))\ s(v)$. The range of v is the elements of the universe of sort D_p for which the guard $g(v)$ is evaluated to *true*, and $v_1,...,v_n$ range over $D_1,...,D_n$, respectively. Suppose $v_1,...,v_n$ and v are transformed into logic program variables $X_1,...,X_n$ and X, respectively. Then the guard $g(v)$ and the body $s(v)$ are transformed into the atoms $g(X_1,...,X_n,X,i)$ and $s(X_1,...,X_n,X,i)$, defined by the logic rules

$g(X_1,...,X_n,X,i) \leftarrow E_t(g(v),i)$, <*side atoms*>, $d_1(X_1)$, ..., $d_n(X_n)$, $d_p(X)$
$s(X_1,...,X_n,X,i) \leftarrow E_t(s(v),i)$, <*side atoms*>, $d_1(X_1)$, ..., $d_n(X_n)$, $d_p(X)$.

The term $(\exists v{:}g(v))\ s(v)$ is transformed into the atom $p(X_1,...,X_n,i)$, which is defined to be the conjunction of $g(X_1,...,X_n,X,i)$ and $s(X_1,...,X_n,X,i)$:

$p(X_1,...,X_n,i) \leftarrow g(X_1,...,X_n,X,i),\ s(X_1,...,X_n,X,i),\ d_1(X_1),\ ...,\ d_n(X_n),\ d_p(X)$.

A first-order term $(\forall v{:}g(v))\ s(v)$ is treated by transforming its equivalence $\neg((\exists v{:}g(v))\ \neg s(v))$.

Transition Rules. Let the transformation of transition rules be a function E_r, which takes a transition rule as a parameter and returns a set of logic rules.

- A *skip rule* does nothing, so $E_r(\texttt{skip})$ is the empty set:

  $E_r(\texttt{skip}) = \emptyset$.

- An *update rule* is transformed into two logic rules, one capturing the result of the firing of the update and the other representing that the left-hand side of the update is updated:

  $E_r(f(t_1,...,t_n)\ \texttt{:=}\ t_p) =$
  { $f(E_t(t_1,I),...,E_t(t_n,I),E_t(t_p,I),I{+}1) \leftarrow$ <*side atoms*>, $has\_next\_state(I)$
  $f\_upd(E_t(t_1,I),...,E_t(t_n,I),I) \leftarrow$ <*side atoms*>, $has\_next\_state(I)$ } .

- A *block rule* is transformed into the union of the sets of logic rules for all the sub-rules:

  $E_r(\texttt{block}\ R_1...R_n\ \texttt{endblock}) = E_r(R_1) \cup ... \cup E_r(R_n)$.

- A *conditional rule* of the form `if` g `then` R_1 `else` R_2 `endif` is transformed as follows. First, create a logic rule whose head is true iff the guard g is *true*. Let the head of the logic rule be $g(I)$. Then add $g(I)$ to the body of each rule in $E_r(R_1)$ and *not* $g(I)$ to the body of each rule in $E_r(R_2)$. In the following, $E_r(R)[+l]$ denotes the set of logic rules obtained by adding the literal l to the body of each rule in $E_r(R)$:

  $E_r(\texttt{if}\ g\ \texttt{then}\ R_1\ \texttt{else}\ R_2\ \texttt{endif}) =$
  { $g(I) \leftarrow E_t(g,I)$, <*side atoms*>, $has\_next\_state(I)$ } $\cup$
  $E_r(R_1)[+g(I)] \cup$
  $E_r(R_2)[+(not\ g(I))]$.

- A *do-forall rule* of the form `do forall` v `:` $g(v)$ R `enddo` is transformed as follows. The range of the head variable v is the elements of the universe of sort D_p for which the guard $g(v)$ is *true*, and we assume that v is transformed into a logic program variable X:

  $E_r(\texttt{do forall}\ v\ \texttt{:}\ g(v)\ R\ \texttt{enddo}) =$
  { $g(X,I) \leftarrow E_t(g(v),I)$, <*side atoms*>, $d_p(X)$, $has\_next\_state(I)$ } $\cup$
  $(E_r(R)[+g(X,I)])[+d_p(X)]$.

Example. Consider a light that is switched on when it is off and switched off when it is on. The following ASM captures this behaviour:

LIGHT = `if` *light* `=` *off* `then` *light* `:=` *on* `else` *light* `:=` *off* `endif`

Let the signature of LIGHT have the sort *Status* with the set of constants $\{on, off\}$ and the controlled nullary function *light*: $\rightarrow$ *Status*, which is *off* in every initial state of LIGHT. The ASM LIGHT is transformed into the following logic program:

$boolean(1)$
$boolean(0)$
$status(on)$
$status(off)$
$light(off,0)$
$light(X,I{+}1) \leftarrow not\ light\_upd(I),\ light(X,I),\ status(X),\ has\_next\_state(I)$
$g(I) \leftarrow light(off,I),\ has\_next\_state(I)$
$light(on,I{+}1) \leftarrow g(I),\ has\_next\_state(I)$
$light\_upd(I) \leftarrow g(I),\ has\_next\_state(I)$
$light(off,I{+}1) \leftarrow not\ g(I),\ has\_next\_state(I)$
$light\_upd(I) \leftarrow not\ g(I),\ has\_next\_state(I).$

Note that we use '1' and '0' in the program for *true* and *false*, respectively.

4.2 Constructing Program $\Pi(\neg\psi,k)$

Recall that a counterexample may have a loop. The existence of a loop is indicated by the equivalence between the last state s_k in the prefix and some previous state s_i, where $0 \leq i \leq k$-1. Two state s_1 and s_2 of M are equivalent iff the values of all locations in s_1 and s_2 are identical.

Let h be a formula that is equivalent to $\neg\psi$ and is in negation normal form. The method to construct the program $\Pi(\neg\psi,k)$, which is derived from [9], is given as follows:

- A counterexample has at most one loop. Let the atom $el(i)$ be in an answer set iff state s_k is equivalent to s_i, where $0 \leq i \leq k$-1, and add the rule

$$0 \leq \{el(0), ..., el(k-1)\} \leq 1.$$

- For each controlled or monitored function f: $D_1 \times ... \times D_n \rightarrow D_p$ in Σ_M, let the atom $el\_f(c_1,...,c_n,i)$ be in an answer set iff $Val_{s_i,\zeta}(f(c_1,...,c_n))$ is equal to $Val_{s_k,\zeta}(f(c_1,...,c_n))$, where $0 \leq i \leq k$-1 and c_j is a constant of D_j for $1 \leq j \leq n$. Add the logic rule

$$\begin{aligned} el\_f(X_1,...,X_n,I) \leftarrow\ & f(X_1,...,X_n,X_p,I),\ f(X_1,...,X_n,X_p,k),\\ & d_1(X_1),\ ...,\ d_n(X_n),\ d_p(X_p),\ has\_next\_state(I). \end{aligned}$$

- For each controlled or monitored function f: $D_1 \times ... \times D_n \rightarrow D_p$, if $el(i)$ is in an answer set, then each possible $el\_f(c_1,...,c_n,i)$ must also be in the answer set. Therefore, add the constraint

$\leftarrow el(I),\ not\ el\_f(X_1,...,X_n,I),\ d_1(X_1),\ ...,\ d_n(X_n),\ has\_next\_state(I).$

- Define the atoms l, $nl(i)$ and $il(i)$, where l is in an answer set iff a loop exists, $nl(i)$ is in an answer set iff state s_i is the 'next state' of the last state, and $il(i)$ is in an answer set iff state s_i is inside the loop (see Fig. 2 for an example). Add the rules

$l \leftarrow el(I),\ has\_next\_state(I)$ $\qquad nl(I+1) \leftarrow el(I),\ has\_next\_state(I)$
$il(I+1) \leftarrow el(I),\ has\_next\_state(I)$ $\qquad il(I+1) \leftarrow il(I),\ has\_next\_state(I).$

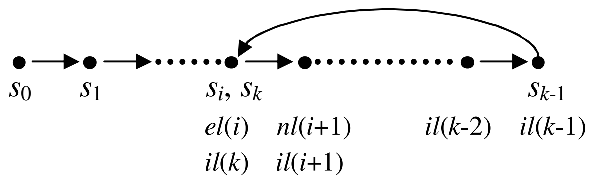

Fig. 2. Loop-related atoms.

- Recursively translate the formula h by adding the rules in Table 1. In the end, we require that a prefix of a run in M satisfy h by including the constraint

$\leftarrow not\ h(0).$

In Table 1, an atomic proposition is an equality of the form $f(c_1,...,c_n)=c_p$, where f is an n-ary function and all c_j are constants for j=1,...,n and p. An equality such as $f_1(c_{11},...,c_{1n})=f_2(c_{21},...,c_{2n})$ can be written as a disjunction of the conjunctions $(f_1(c_{11},...,c_{1n})=c_l \wedge f_2(c_{21},...,c_{2n})=c_l)$. This is correct because in $\Pi(M,k)$ each sort D_p in Σ_M is represented as a domain predicate d_p whose extent is defined as a finite set of constants. The program $\Pi(M,k)$ restricts that the range of a function of D_p must be the values of the constants $\{c_{p1},...,c_{pl}\}$ for which there is a rule $d_p(c_{pj})$ in $\Pi(M,k)$ for $1 \leq j \leq l$.

To illustrate the construction of $\Pi(\neg\psi,k)$, we let the formula h_1 be the proposition $light{=}on$ and h be the formula $\mathbf{F}h_1$. Then h is translated into the following logic program:

$0 \leq \{el(0),...,el(k-1)\} \leq 1$
$el\_light(I) \leftarrow el\_light(X,I),\ el\_light(X,k),\ status(X),\ has\_next\_state(I)$
$\leftarrow el(I),\ not\ el\_light(I),\ has\_next\_state(I)$
$l \leftarrow el(I),\ has\_next\_state(I)$
$nl(I+1) \leftarrow el(I),\ has\_next\_state(I)$
$il(I+1) \leftarrow el(I),\ has\_next\_state(I)$
$il(I+1) \leftarrow il(I),\ has\_next\_state(I)$
$h_1(I) \leftarrow light(on,I),\ valid\_state(I)$
$h(I) \leftarrow h_1(I),\ valid\_state(I)$
$h(I) \leftarrow h(I+1),\ valid\_state(I)$
$h(k+1) \leftarrow nl(I),\ h(I),\ valid\_state(I).$

Formula h	Translation
atomic proposition $f(c_1,...,c_n){=}c_p$	$h(I) \leftarrow f(c_1,...,c_n,c_p)$, $valid\_state(I)$ if f is static $h(I) \leftarrow f(c_1,...,c_n,c_p,I)$, $valid\_state(I)$ otherwise
$\neg(f(c_1,...,c_n){=}c_p)$	$h(I) \leftarrow not\ f(c_1,...,c_n,c_p)$, $valid\_state(I)$ if f is static $h(I) \leftarrow not\ f(c_1,...,c_n,c_p,I)$, $valid\_state(I)$ otherwise
$h_1 \wedge h_2$	$h(I) \leftarrow h_1(I),\ h_2(I),\ valid\_state(I)$
$h_1 \vee h_2$	$h(I) \leftarrow h_1(I),\ valid\_state(I)$ $h(I) \leftarrow h_2(I),\ valid\_state(I)$
$\mathbf{G}h_1$	$h(I) \leftarrow h_1(I),\ h(I{+}1),\ valid\_state(I)$ $h(k{+}1) \leftarrow l,\ not\ q$ $q \leftarrow il(I),\ not\ h_1(I),\ valid\_state(I)$
$\mathbf{F}h_1$	$h(I) \leftarrow h_1(I),\ valid\_state(I)$ $h(I) \leftarrow h(I{+}1),\ valid\_state(I)$ $h(k{+}1) \leftarrow nl(I),\ h(I),\ valid\_state(I)$
$\mathbf{X}h_1$	$h(I) \leftarrow h_1(I{+}1),\ has\_next\_state(I)$ $h(k) \leftarrow nl(I),\ h_1(I),\ valid\_state(I)$

Table 1. Translation of an LTL formula.

4.3 Correctness of the Method

If the program $\Pi(M,k) \cup \Pi(\neg\psi,k)$ has an answer set, then there is in M a prefix of a run that satisfies $\neg\psi$. Such an answer set corresponds to a counterexample for ψ. We formally state this result as the following theorem:

Theorem 1. If $\Pi(M,k) \cup \Pi(\neg\psi,k)$ has an answer set A, then there exists in M a prefix $(s_0,...,s_k)$ of a run satisfying $\neg\psi$ such that for each controlled or monitored function f: $D_1 \times ... \times D_n \rightarrow D_p$ in Σ_M and $0 \leq i \leq k$, $Val_{s_i,\zeta}(f(c_1,...,c_n)) = Val_{s_i,\zeta}(c_p)$ iff the atom $f(c_1,...,c_n,c_p,i)$ is in A, where c_j is a constant of sort D_j for $j{=}1,...,n$ and p.

We say that the prefix $(s_0,...,s_k)$ is *described* by the answer set A. The answer set A may have atoms that are not of the form $f(c_1,...,c_n,c_p,i)$. However, to obtain a prefix from A, only the atoms of the form $f(c_1,...,c_n,c_p,i)$ are needed. The reason is that a state s of M is identified by the values of all the locations in s. As indicated by Theorem 1, the values of the locations can be obtained from the atoms of the form $f(c_1,...,c_n,c_p,i)$ in A.

An outline of the proof of Theorem 1 is given in Appendix.

5 Experiments

Based on the method explained in the previous section, we have implemented the automated transformation of ASM models and LTL formulas into logic programs. The input ASM language is ASM-SL [3]. The reason for the choice of ASM-SL is that we want to compare our method to SAT-based BMC of ASMs with

NuSMV. NuSMV is an extension of the BDD-based model checker SMV [13], and a translator from ASM-SL to the language of SMV has been developed by Winter [18]. Therefore, it is logical for our tool to accept the same ASM syntax as Winter's translator. Currently our tool only handles a subset of ASM-SL, but it is sufficient for the purpose of conducting our experiments. Enhancing our tool to cover the full ASM-SL is a possible direction of future work.

For our experiments, we chose the FLASH Cache Conherence Protocol used by Winter as a case study in [18]. The protocol supports sharing of distributed memory among interconnected processors. Winter listed three properties for which counterexamples were found. Based on the explanation in [18], we made the necessary changes to Winter's correct ASM model in order to reproduce the errors she detected when checking the three properties. The purpose was to create counterexample scenarios. We translated the modified ASM into a logic program and a NuSMV model and formulated the properties in LTL[2].

We compared the answer set solvers `SMODELS` and `CMODELS` [11] to the BMC tool in NuSMV. `CMODELS` uses SAT solvers to find answer sets and also requires `LPARSE` as a front-end. The version of NuSMV we used was the same as in [10], and the SAT solver invoked by both `CMODELS` and NuSMV was zChaff [14]. We followed the industrial practice of BMC by starting with a small time bound at 10. Then we increased it by 2 until a counterexample was found or no result was given within the limit of 3600 seconds. When the limit was exceeded, we recorded the time taken to prove the absence of counterexamples within the previous bound. The results are given in Table 2. The first column lists the properties, named P1, P2 and P3, respectively. In the other columns, each entry reports the total time needed to find a counterexample (written as 'k=n, t', where n is the value of the bound k and t is the time in seconds) or prove the absence of counterexamples within the bound (written as 'k>n, t').

Property	`SMODELS`	`CMODELS`	NuSMV(BMC)
P1	k>12, 244.5	k=18, 16.33	k=18, 2387.63
P2	k>10, 13.03	k=16, 71.76	k=16, 1972.69
P3	k>10, 8.28	k>24, 2847.83	k>18, 2392.45

Table 2. BMC Experiments for FLASH Cache Coherence Protocol.

As expected, counterexamples were found for properties P1 and P2. None were found for P3 as we had too little information to change Winter's model to falsify it. For all three properties, `CMODELS` outperformed NuSMV. The former was more than 146 times faster for P1 and was able to reach a higher bound for P3 without exceeding the time limit. `SMODELS` failed to find a counterexample for P1 and P2. For P2 and P3, it managed to terminate within the limit for only the

[2] The ASM model, LTL formulas, logic programs, NuSMV models and tools used in the experiments are available at `http://www.sfu.ca/~ctang/experiments.html`.

initial bound 10. Further investigation is needed to see whether our automatically generated logic program and the NuSMV code produced by Winter's translator are optimal as they can have a significant impact on the performances.

6 Related Work

Several ASM model checking methods based on existing model checkers are available in the literature. In [18], Winter showed how to transform ASMs into the syntax of SMV. In her algorithm, every location that appears as a parameter of a function not supported by the model checker must be unfolded into every possible value. With the use of logic program variables in our approach, location unfolding is handled by the answer set tool during the process of grounding. Winter also proposed an encoding of ASMs in the language MDG-HDL accepted by the model checker MDG. MDG is based on Multiway Decision Graphs, which subsume BDDs, and supports abstract data types and uninterpreted functions. A case study of the MDG approach can be found in [8]. In [7], the authors presented a translation from the ASM language PROMELA to the language of the model checker Spin. Spin is an explicit state model checker in which reachable states are computed by executing the model rather than being represented as boolean functions. None of the above-mentioned work discusses BMC of ASMs, although Winter's transformation algorithm for SMV allows BMC of ASMs using NuSMV.

With respect to the application of ASP to model checking, Heljanko and Niemelä introduced in [9] bounded LTL model checking of 1-safe Petri nets with ASP. Their approach is similar to ours in that 1-safe Petri nets and LTL formulas are translated into logic programs such that BMC problems become tasks of finding answer sets. A notable difference between the two approaches, other than different model representations, is that ours requires a much more extensive use of variables in logic programs in order to encode terms in ASMs.

7 Conclusions

We present the application of answer set programming to bounded LTL model checking of abstract state machines. We introduce a mapping from a BMC problem for an ASM to a problem of answer set computation through a compact encoding of the ASM and LTL formula in a logic program. The compact encoding is realized by the power of logic rules to succinctly represent the transition behaviour of ASMs and the recursive evaluation of LTL formulas. Initial experimental results show that the application of ASP is promising in BMC of ASMs. We are looking to conduct more experiments to further understand the capability of our method and perhaps find ways to improve it.

References

1. A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. *Bounded Model Checking*, volume 58 of *Advances In Computers*. Elsevier, 2003.

2. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003.
3. G. Del Castillo. *ASM-SL, a Specification Language based on Gurevich's Abstract State Machines: Introduction and Language Definition*. Heinz Nixdorf Institut Paderborn, Germany, 1999.
4. W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24:498–520, 1998.
5. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Proceedings of International Conference on Computer-Aided Verification*, 2002.
6. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
7. A. Gargantini, E. Riccobene, and S. Rinzivillo. Using Spin to generate tests from asm specifications. *Lecture Notes in Computer Science*, 2589:263–277, 2003.
8. A. Gawanmeh, S. Tahar, and K. Winter. Interfacing ASM with the MDG tool. *Lecture Notes in Computer Science*, 2589:278–292, 2003.
9. K. Heljanko and I. Niemelä. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming*, 3:519–550, 2003.
10. T. Latvala, A. Biere, K. Heljanko, and T. Junttila. Simple is better: Efficient bounded model checking for past LTL. In Radia Cousot, editor, *Verification, Model Checking and Abstract Interpretation 2005, 6th International Conference VMCAI'05, Paris, France*, 2005.
11. Y. Lierler and M. Maratea. Cmodels-2: Sat-based answer sets solver enhanced to non-tight programs. In *Proceedings of the 7th International Conference on Logic Programming and Non-monotonic Reasoning*, 2004.
12. V. Lifschitz. Introduction to answer set programming, 2004. Introductory course at the 16th European Summer School in Logic, Language and Information.
13. K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
14. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficent SAT solver. In *Proceedings of the 38th Design Automation Conference*, 2001.
15. I. Niemelä and P. Simons. Extending the Smodels system with cardinality and weight constraints. In *Logic-Based Artificial Intelligence*. Kluwer Academic Publishers, 2000.
16. T. Syrjänen. *Lparse 1.0 User's Manual available at `http://www.tcs.hut.fi/Software/smodels/lparse.ps`*, 2000.
17. J. Wing and M. Vaziri-Farahani. A case study in model checking software. *Science of Computer Programming*, 28:273–299, 1997.
18. K. Winter. *Model Checking Abstract State Machines*. PhD thesis, Technical University of Berlin, 2001.

Appendix: Outline of Proof of Theorem 1

We divide the proof of Theorem 1 into the proofs of the following two statements:

1. If $\Pi(M,k) \cup \Pi(\neg\psi,k)$ has an answer set A, there is in M a prefix $(s_0,\ldots,s_k)$ of a run such that for each controlled or monitored function $f\colon D_1 \times \ldots \times D_n \rightarrow D_p$ in Σ_M and $0 \leq i \leq k$, $Val_{s_i,\zeta}(f(c_1,\ldots,c_n)) = Val_{s_i,\zeta}(c_p)$ iff the atom $f(c_1,\ldots,c_n,c_p,i)$ is in A, where c_j is a constant of sort D_j for $j=1,\ldots,n$ and p.

2. The prefix $(s_0,...,s_k)$ satisfies $\neg\psi$.

To prove the statement 1, we use the following proposition given in [9]:

Proposition 1. Let Π_1 and Π_2 be two programs such that the atoms in the heads of the rules in Π_2 do not appear in Π_1. Then for each answer set A of $\Pi_1 \cup \Pi_2$, $A \cap Atoms(\Pi_1)$ is an answer set of Π_1, where $Atoms(\Pi_1)$ is the set of atoms that occur in Π_1.

We place the two logic rules which define the domain predicate *valid_state* and *has_next_state* in $\Pi(M,k)$. Then the atoms in the heads of the rules in $\Pi(\neg\psi,k)$ do not occur in $\Pi(M,k)$. By Proposition 1, if A is an answer set of $\Pi(M,k) \cup \Pi(\neg\psi,k)$, then $A(M,k)=A \cap Atoms(\Pi(M,k))$ is an answer set of $\Pi(M,k)$. Therefore, to prove the statement 1, it is sufficient to show that there is in M a prefix $(s_0,...,s_k)$ of a run such that for each controlled or monitored function f and $0 \leq i \leq k$, $Val_{s_i,\zeta}(f(c_1,...,c_n))=Val_{s_i,\zeta}(c_p)$ iff $f(c_1,...,c_n,c_p,i) \in A(M,k)$. This can be proved by induction on i.

For i=0, if $f(c_1,...,c_n,c_p,0) \in A(M,k)$, then one of the following two rules must be in $\Pi(M,k)$:

$$f(c_1,...,c_n,c_p,0) \qquad 1 \leq \{f(c_1,...,c_n,c_{p1},0),...,f(c_1,...,c_n,c_{pl},0)\} \leq 1.$$

In either case, we can argue that there exists some initial state s_0 of M such that $Val_{s_0,\zeta}(f(c_1,...,c_n))=Val_{s_0,\zeta}(c_p)$ for each f. Then we assume that $f(c_1,...,c_n,c_p,0) \notin A(M,k)$ but $Val_{s_0,\zeta}(f(c_1,...,c_n))=Val_{s_0,\zeta}(c_p)$. In this case, an atom $f(c_1,...,c_n,c_{p'},0)$ must be in $A(M,k)$ for some constant $c_{p'}$ with a different interpretation from c_p due to the two rules above with c_p replaced by $c_{p'}$ in the first one. Then we have $Val_{s_0,\zeta}(f(c_1,...,c_n))=Val_{s_0,\zeta}(c_{p'})$, a contradiction to our assumption that $Val_{s_0,\zeta}(f(c_1,...,c_n))=Val_{s_0,\zeta}(c_p)$. Therefore, $Val_{s_0,\zeta}(f(c_1,...,c_n))=Val_{s_0,\zeta}(c_p)$ iff $f(c_1,...,c_n,c_p,0) \in A(M,k)$ for each f.

For the inductive hypothesis, we assume that for all $i<k$, there exists in M a prefix $(s_0,...,s_i)$ of a run such that $Val_{s_i,\zeta}(f(c_1,...,c_n))=Val_{s_i,\zeta}(c_p)$ iff $f(c_1,...,c_n,c_p,i) \in A(M,k)$ for each controlled or monitored function f. We need to prove that the prefix can be extended to $(s_0,...,s_{i+1})$ such that $Val_{s_{i+1},\zeta}(f(c_1,...,c_n))=Val_{s_{i+1},\zeta}(c_p)$ iff $f(c_1,...,c_n,c_p,i+1) \in A(M,k)$ for each f. A monitored function f changes nondeterministically, and so the prefix can definitely be extended to $(s_0,...,s_{i+1})$ such that for each monitored function f, $Val_{s_{i+1},\zeta}(f(c_1,...,c_n))=Val_{s_{i+1},\zeta}(c_p)$ if $f(c_1,...,c_n,c_p,i+1) \in A(M,k)$. Then we assume that $f(c_1,...,c_n,c_p,i+1) \notin A(M,k)$ but $Val_{s_{i+1},\zeta}(f(c_1,...,c_n))=Val_{s_{i+1},\zeta}(c_p)$. Because f is monitored, the following rule is in $\Pi(M,k)$:

$$1 \leq \{\, f(X_1,...,X_n,c_{p1},I+1),...,f(X_1,...,X_n,c_{pl},I+1) \,\} \leq 1 \leftarrow \\ d_1(X_1),\ ...,\ d_n(X_n),\ has\_next\_state(I).$$

Based on our assumption and the rule above, we can reach a contradiction similar to the one in the base case. Therefore, $Val_{s_{i+1},\zeta}(f(c_1,...,c_n))=Val_{s_{i+1},\zeta}(c_p)$ iff $f(c_1,...,c_n,c_p,i+1) \in A(M,k)$ for each monitored function f.

If f is a controlled function, and $f(c_1,...,c_n,c_p,i{+}1){\in}A(M,k)$, exactly one of the following two cases holds:

1. An update rule $f(t_1,...,t_n)$ `:=` t_p in M results in the following logic rule in $\Pi(M,k)$:

   $$f(E_t(t_1,I),...,E_t(t_n,I),E_t(t_p,I),I{+}1) \leftarrow \textit{<side atoms>},\ \textit{has\_next\_state}(I). \quad (5)$$

   Potentially there may be other literals added to the body of the rule (5) as the update rule may be a sub-rule in a conditional or do-forall rule. We just consider unconditional updates for now. The grounding of the rule (5) leads to a rule with $f(c_1,...,c_n,c_p,i{+}1)$ as the head (recall grounding is the process of substituting values for variables):

   $$f(c_1,...,c_n,c_p,i{+}1) \leftarrow \textit{<side atoms>},\ \textit{has\_next\_state}(i). \quad (6)$$

   The body of the rule (6) is satisfied, which is why $f(c_1,...,c_n,c_p,i{+}1){\in}A(M,k)$. We want to prove that the update $((f,(Val_{s_i,\zeta}(c_1),..., Val_{s_i,\zeta}(c_n))), Val_{s_i,\zeta}(c_p))$ is in the update set in s_i. In order to do so, we need to establish that $Val_{s_i,\zeta}(t_j){=}Val_{s_i,\zeta}(c_j)$ for $j{=}1,...,n$ and p, where t_j is a term of sort D_j and c_j is a constant of D_j. We can do so by considering the different cases for terms. The inductive hypothesis is applied if t_j is a function application. For instance, if t_j is $g(c_{j1},...,c_{jm})$, where g is an m-ary function and $c_{j1},...,c_{jm}$ are constants, then $E_t(t_j,i)$ is a logic program variable X_j. If g is static, the side atoms are $g(c_{j1},...,c_{jm},X_j)$ and $d_j(X_j)$. In the rule (6), X_j is replaced with the constant c_j. Because the body of the rule (6) is satisfied, the atom $g(c_{j1},...,c_{jm},c_j)$ must also be satisfied. This indicates that the rule $g(c_{j1},...,c_{jm},c_j)$ is in $\Pi(M,k)$ and thus $Val_{s,\zeta}(g(c_{j1},...,c_{jm})){=}Val_{s,\zeta}(c_j)$ for all s and ζ. Therefore, $Val_{s_i,\zeta}(t_j){=}Val_{s_i,\zeta}(c_j)$. If g is controlled or monitored, the side atoms are $g(c_{j1},...,c_{jm},X_j,i)$ and $d_j(X_j)$. In the rule (6), the atom $g(c_{j1},...,c_{jm},c_j,i)$ must be satisfied and hence in $A(M,k)$. By the inductive hypothesis, $Val_{s_i,\zeta}(t_j){=}Val_{s_i,\zeta}(c_j)$.
   Because the update rule $f(t_1,...,t_n)$ `:=` t_p is not guarded, and $Val_{s_i,\zeta}(t_j){=}Val_{s_i,\zeta}(c_j)$ for $j{=}1,...,n$ and p, the update $((f,(Val_{s_i,\zeta}(c_1),..., Val_{s_i,\zeta}(c_n))), Val_{s_i,\zeta}(c_p))$ must be in the update set in s_i. The firing of the update set in s_i extends the prefix to $(s_0,...,s_{i+1})$ such that $Val_{s_{i+1},\zeta}(f(c_1,...,c_n)){=}Val_{s_{i+1},\zeta}(c_p)$.
   If the update rule $f(t_1,...,t_n)$ `:=` t_p is a sub-rule in a conditional or do-forall rule, other literals will be added to the body of the rule (5). If the update rule is in the then part of a conditional rule with the guard g, the atom $g(I)$ is added to the body of the rule (5), and the logic rule

   $$g(I) \leftarrow E_t(g,I),\ \textit{<side atoms>},\ \textit{has\_next\_state}(I) \quad (7)$$

   supports $g(I)$, which does not occur as a head of any other rules in the program. The grounding of the rule (5) makes the atom $g(i)$ appear in the body of the rule (6). Because the body of the rule (6) is satisfied, $g(i)$ must be satisfied. Therefore, the body of some rule with $g(i)$ as the head is satisfied, and it must be a conjunction of literals from the grounding of the body of the rule (7). Then we can argue that $Val_{s_i,\zeta}(g)=Val_{s_i,\zeta}(true)$ in the same way as we do for $Val_{s_i,\zeta}(t_j)=Val_{s_i,\zeta}(c_j)$ for j=1,...,n and p, and thus the update $((f,(Val_{s_i,\zeta}(c_1),...,Val_{s_i,\zeta}(c_n))),Val_{s_i,\zeta}(c_p))$ must be in the update set in s_i. The argument is similar for an update rule that is in the else part of a conditional rule or in a do-forall rule.

2. The atom $f(c_1,...,c_n,c_p,i{+}1)$ is in $A(M,k)$ because the body of the following rule is satisfied:

$$f(c_1,...,c_n,c_p,i{+}1) \leftarrow \mathit{not}\ f\_upd(c_1,...,c_n,i),\ f(c_1,...,c_n,c_p,i),\\ d_1(c_1),\ ...,\ d_n(c_n),\ d_p(c_p),\ has\_next\_state(i). \tag{8}$$

The rule (8) results from the grounding of the following rule in $\Pi(M,k)$:

$$f(X_1,...,X_n,X_p,I{+}1) \leftarrow \mathit{not}\ f\_upd(X_1,...,X_n,I),\ f(X_1,...,X_n,X_p,I),\\ d_1(X_1),\ ...,\ d_n(X_n),\ d_p(X_p),\ has\_next\_state(I).$$

The atom $f\_upd(c_1,...,c_n,i)$ is not satisfied, which means that the bodies of all rules of the form

$$f\_upd(c_1,...,c_n,i) \leftarrow \langle side\ atoms\rangle,\ has\_next\_state(i),\ \langle literals\rangle \tag{9}$$

are not satisfied, where <*literals*> denotes the set of literals added to the body of the rule. Therefore, the bodies of all rules of the form

$$f(c_1,...,c_n,c,i{+}1) \leftarrow \langle side\ atoms\rangle,\ has\_next\_state(i),\ \langle literals\rangle \tag{10}$$

are not satisfied for all constants c. The reason is that with the way an update rule is translated into logic rules, for each rule LR_1 of the form (10) in $\Pi(M,k)$, there is a rule LR_2 of the form (9) such that the set of side atoms in LR_2 must be a subset of that in LR_1. The result that the bodies of all rules of the form (10) are not satisfied for all constants c allows us to show that no update $((f,(Val_{s_i,\zeta}(c_1),...,Val_{s_i,\zeta}(c_n))),Val_{s_i,\zeta}(c))$ is in the update set in s_i for any constant c. In addition, $f(c_1,...,c_n,c_p,i)$ is satisfied. By the inductive hypothesis, $Val_{s_i,\zeta}(f(c_1,...,c_n))=Val_{s_i,\zeta}(c_p)$. Therefore, by the semantics of ASMs, $Val_{s_{i+1},\zeta}(f(c_1,...,c_n))=Val_{s_{i+1},\zeta}(c_p)$ after the firing of the update set in s_i.

For the other direction, again we try to reach a contradiction by assuming that $f(c_1,...,c_n,c_p,i{+}1)\notin A(M,k)$ but $Val_{s_{i+1},\zeta}(f(c_1,...,c_n))=Val_{s_{i+1},\zeta}(c_p)$. We first show that there must be an atom $f(c_1,...,c_n,c_{p'},i{+}1)$ in $A(M,k)$ for some constant $c_{p'}$ with a different interpretation from c_p by assuming the opposite is true. Then if $f\_upd(c_1,...,c_n)$ is not satisfied, the bodies of all rules of the form

$$f(c_1,...,c_n,c,i{+}1) \leftarrow \mathit{not}\ f\_upd(c_1,...,c_n,i),\ f(c_1,...,c_n,c,i),\\ d_1(c_1),\ ...,\ d_n(c_n),\ d_p(c),\ has\_next\_state(i).$$

are not satisfied for all constants c of sort D_p. This means that $f(c_1,...,c_n,c,i)$ are not satisfied and hence not in $A(M,k)$ for all constants c. We take i=0 so that $f(c_1,...,c_n,c,0)\notin A(M,k)$ for all c. However, this is not possible because in $\Pi(M,k)$ there must be a rule $f(c_1,...,c_n,c,0)$ for some c or the rule 1 $\leq$

$\{f(c_1, ..., c_n, c_{p1}, 0), ..., f(c_1, ..., c_n, c_{pl}, 0)\} \leq 1$. Consequently, the assumption that $f(c_1,...,c_n,c,i{+}1){\notin}A(M,k)$ for all c is incorrect. If $f\_upd(c_1,...,c_n)$ is satisfied, then there must be a rule LR of the form (9) in $\Pi(M,k)$ whose body is satisfied. In addition, there are rules $LR_1,...,LR_l$ of the form (10) such that the set of the side atoms in LR is a subset of that in each of $LR_1,...,LR_l$. The bodies of $LR_1,...,LR_l$ are not satisfied by the assumption that $f(c_1,...,c_n,c,i{+}1){\notin}A(M,k)$ for all c. Based on this result, we can show that for some m-ary function g of sort D_q and all constants c' of D_q, all atoms $g(c_{1'},...,c_{m'},c',i)$ (or $g(c_{1'},...,c_{m'},c')$ if g is static) are not in $A(M,k)$ for some $c_{1'},...,c_{m'}$. Each of these atoms is a side atom in some of $LR_1,...,LR_l$. We let i be 0 so that $g(c_{1'},...,c_{m'},c',0){\notin}A(M,k)$ (or $g(c_{1'},...,c_{m'},c'){\notin}A(M,k)$) for all c', which is not possible. Therefore, there must be an atom $f(c_1,...,c_n,c_{p'},i{+}1)$ in $A(M,k)$ for some constant $c_{p'}$ with a different interpretation from c_p. This means that $Val_{s_{i+1},\zeta}(f(c_1,...,c_n)){=}Val_{s_{i+1},\zeta}(c_{p'})$, which contradicts our assumption that $Val_{s_{i+1},\zeta}(f(c_1,...,c_n)){=}Val_{s_{i+1},\zeta}(c_p)$.

We have completed the outline of the proof of the statement 1. The proof of the statement 2 is derived from [9], in which the following proposition is used:

Proposition 2. Let Π_1 be a program and Π_2 be a stratified program such that the atoms in the heads of the rules in Π_2 do not appear in Π_1. Then for each answer set A_1 of Π_1, there is a unique answer set A of $\Pi_1{\cup}\Pi_2$ such that $A_1{=}A{\cap}Atoms(\Pi_1)$.

A program is *stratified* if its dependency graph has no cycles with arcs labelled with '$\neg$'. In a dependency graph of a program Π, the nodes are the atoms in Π and there is a directed arc from p to q if there is a rule in Π with q in the head and p in the body. The arc is labelled with '$\neg$' if p is negated.

Let h be an LTL formula that is equivalent to $\neg\psi$ and is in negation normal form. We divide $\Pi(\neg\psi,k)$ into three programs: $\Pi_1(\neg\psi,k)$, which consists of the rules defining all loop-related atoms, $\Pi_2(\neg\psi,k)$, which has the translation of h as given in Table 1, and $\Pi_3(\neg\psi,k)$, which is the constraint $\leftarrow$ *not* $h(0)$. The following three properties hold:

1. The program $\Pi_2(\neg\psi,k)$ is stratified.
2. The atoms that occur as heads in $\Pi_2(\neg\psi,k)$ do not appear in $\Pi(M,k){\cup}\Pi_1(\neg\psi,k)$.
3. Exactly one of the following two cases is true:
   (a) $nl(j{+}1){\in}A_1$, $nl(i){\notin}A_1$ for all $i{\neq}j{+}1$, $l{\in}A_1$, and $il(i){\in}A_1$ for all $j{+}1 \leq i \leq k$.
   (b) $nl(i{+}1){\notin}A_1$ for all $0 \leq i \leq k$-1 and $l{\notin}A_1$.

Based on the properties 1 and 2 along with Proposition 2, if $\Pi(M,k){\cup}\Pi_1(\neg\psi,k)$ has an answer set A_1, then $\Pi(M,k){\cup}\Pi_1(\neg\psi,k){\cup}\Pi_2(\neg\psi,k)$ has an answer set A_2. Let the prefix $(s_0,...,s_k)$ of a run described by A_2 be π. We need to demonstrate that the translation of h in $\Pi_2(\neg\psi,k)$ respects the semantics of LTL, i.e. $\pi_i \models h$ iff $h(i){\in}A_2$ for $0 \leq i \leq k$, by induction on the structure of the formula h. That is, if h_1 is a sub-formula of h, we assume that the translation of h_1 respects the semantics of LTL.

If h is a proposition $f(c_1,...,c_n)=c_p$ or its negation $\neg(f(c_1,...,c_n)=c_p)$, then the atom $f(c_1,...,c_n,i)$ (or $f(c_1,...,c_n)$ if f is static) is in A_2 iff $Val_{s_i,\zeta}(f(c_1,...,c_n))=Val_{s_i,\zeta}(c_p)$. If h is $h_1 \wedge h_2$ or $h_1 \vee h_2$, the translation clearly respects the semantics of LTL. If h is $\mathbf{G}h_1$, $\mathbf{F}h_1$ or $\mathbf{X}h_1$, we can show the following based on the property 3:

– If the translation of h respects the semantics of LTL for $i+1$, then it does for i, where $0 \leq i \leq k$-1, and
– The translation of h respects the semantics of LTL for k.

The proofs for the operators **G**, **F** and **X** are based on the following equivalences for $0 \leq i \leq k$-1:

– $\pi_i \models \mathbf{G}h_1$ iff $\pi_i \models h_1$ and $\pi_{i+1} \models \mathbf{G}h_1$,
– $\pi_i \models \mathbf{F}h_1$ iff $\pi_i \models h_1$ or $\pi_{i+1} \models \mathbf{F}h_1$, and
– $\pi_i \models \mathbf{X}h_1$ iff $\pi_{i+1} \models h_1$.

If the translation of h in $\Pi_2(\neg\psi,k)$ respects the semantics of LTL, then it follows that whenever $\Pi(M,k) \cup \Pi(\neg\psi,k)$ (i.e., $\Pi(M,k) \cup \Pi_1(\neg\psi,k) \cup \Pi_2(\neg\psi,k) \cup \Pi_3(\neg\psi,k)$) has an answer set A, the prefix of a run described by A satisfies h because $h(0)$ must be in A.