Software Verification using the Stable Model Semantics

Ravi Palla, Xiaolong Zhang, Yunsong Meng

December 8, 2009

Project Report

1 Introduction

The goal of software verification is to make sure that a software satisfies all the requirements. The most widely used verification techniques are testing and simulation. In the case of complex, asynchronous systems, however, these techniques can cover only a limited portion of possible behaviors. An alternative verification method is Model Checking. It is mainly used to detect errors that are difficult to uncover through testing and simulation. In this approach, the target system is usually modeled as a finite state transition system, and the specifications are expressed in temporal logic. Then, by exhaustively exploring the state space of the state transition system, we can check automatically if the specifications are satisfied or have some counterexamples. The termination of model checking is guaranteed by the finiteness of the model. One of the most important advantages of model checking is that, a counterexample (i.e., a witness of the offending behavior of the system) is produced when a specification is found to be invalid.

2 Model Checking

Generally, in model checking, the behaviour of the system is represented as a Kripke structure $M$. $M$ is then checked against a system property $j$ (a specification of the system)

- If $M \models j$, then the system satisfies the property $j$.
- If $M \not\models j$, then the system does not satisfy the property $j$ and a counter-example can be retrieved.

\[1\text{The state-less model checking approaches are covered in brief later in this report.}\]
This procedure is depicted in the figure below 2:

Some of the advantages of the model checking approach are given below:

- Given a model of the system and the specifications in temporal logic, this process can be automated.
- The counter-examples produced by the approach are valuable for debugging.
- The approach can be used to catch a large number of bugs and is very useful for verifying concurrency related features.

The approach, however, has a few drawbacks. The number of states in a model of the software system can be very large thereby making the process very slow. Often, the system needs to be simplified (abstraction) in order for the approach to be computationally feasible. Also, reverse engineering of already implemented systems in order to obtain the models often does not work.

2.1 Explicit state vs Symbolic Model Checking

There are 2 main paradigms for model checking: explicit state model checking and symbolic model checking. Given the model of a system, in explicit state model checking, states are enumerated on-the-fly with respect to the execution paths and these states are stored in a hash table in order to verify if the execution paths satisfy the specifications of the system. On the other hand, in symbolic model checking, the transition relation is encoded as a BDD or SAT (or other related problem) instance and models of these instances are used to check if the execution paths satisfy the specifications of the system. The following table compares the

---

Figure from [http://iplu.vtt.fi/digitalo/modelchecking.pdf](http://iplu.vtt.fi/digitalo/modelchecking.pdf) presentation by Keijo Heljanko, Helsinki University of Technology
characteristics of the 2 paradigms:

<table>
<thead>
<tr>
<th>Explicit State</th>
<th>Symbolic model checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory intensive</td>
<td>Can handle large state spaces</td>
</tr>
<tr>
<td>Good for finding concurrency errors</td>
<td>Not as good for asynchronous systems</td>
</tr>
<tr>
<td>Short execution paths are better, but long execution paths can be handled</td>
<td>Cannot deal well with long execution paths</td>
</tr>
<tr>
<td>Can handle dynamic creation of objects/threads</td>
<td>Works best with a static transition relation, hence does not deal well with dynamic creation of objects/threads.</td>
</tr>
<tr>
<td>Mostly used in software</td>
<td>Mostly used in hardware but can be very useful for verifying software systems also.</td>
</tr>
</tbody>
</table>

2.2 Different approaches to Model Checking

The following are some of the approaches to Model Checking:

- Symbolic Model Checking using Binary Decision Diagrams (BDDs):
  The first widely used method of model checking is symbolic model checking [McMillan, 1993], in which states and transitions are represented as boolean functions using Ordered Binary Decision Diagrams (BDDs) [Bryant, 1986]. 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.

- Bounded Model Checking (BMC):
  This approach focuses on the search for counter-examples of bounded length and can often handle much larger designs than BDD-based approaches. In this approach, the state transitions and Linear Temporal Logic (LTL) formulas (representing the specifications) for $k$ steps are encoded as a SAT (or other related problem) instance ([Biere et al., 1999], [Heljanko and Niemela, 2001], [Tang and Ternovska, 2005]) to check if any property is violated in $k$ or fewer steps of execution. The value of $k$ is increased until a counter-example is found or until the problem becomes intractable or until a known upper bound is reached. F-soft 3 is a tool based on this idea.

- Abstraction:
  This approach attempts to prove properties of a system by simplifying it. The simplified system usually does not satisfy exactly the same properties as the original one so that a process of refinement may be necessary. Generally, one requires the abstraction to be sound (the properties proved on the abstraction are true of the original system); however,

most often, the abstraction is not complete (not all true properties of the original system are true of the abstraction).

• State-less Model Checking:
  This approach takes programs as input directly unlike the above mentioned approaches.
  This approach does not store any states and thus needs to limit search-depth to ensure termination.

In this report, we focus on BMC using the Stable Model semantics ([Gelfond and Lifschitz, 1988]).

3 Review of Stable Model Semantics

Stable model semantics, which is the basis of answer set programming (ASP), is based on the idea of "negation as failure" in logic programs. Introduced by M.Gelfond and V.Lifschitz [Gelfond and Lifschitz, 1988], it is well developed and widely accepted, and has many efficient implementations called answer set solvers. The notion of "negation as failure" (not) is different from classical negation in that, not A can be derived if "A" cannot be derived. In [Gelfond and Lifschitz, 1988], the stable model semantics were presented for logic programs consisting of rules of the form

\[ A \leftarrow B_1, B_2, \ldots, B_m, \text{not } B_{m+1}, \ldots, \text{not } B_n \]  

(1)

where A and each B_i are atoms and \(0 \leq m \leq n\). A is the head of the rule and B_1, B_2, \ldots, B_m, not B_{m+1}, \ldots, not B_n is the body. Intuitively, the above rule means that if there is reason to believe B_1, \ldots, B_m and there is no reason to believe any of B_{m+1}, \ldots, B_n, then atom A is true. The models of such programs under the stable model semantics are referred to as stable models or answer sets. This semantics is the basis for the work in [Heljanko and Niemela , 2001] and [Tang and Ternovska, 2005]). The semantics was later extended to allow disjunction in the head. Programs consisting of such rules are called as disjunctive logic programs. CMODELS 4, ClaspD 5 and DLV 6 are some of the answer set solvers that can compute the answer sets of disjunctive logic programs. The stable model semantics was further extended to arbitrary propositional formulas in [Ferraris, 2005] and to first-order sentences in [Ferraris et al., 2007]. In this report we use the semantics presented in the latter. We review the semantics below.

The stable models in [Ferraris et al., 2007] are defined in terms of SM operator: given a first-order sentence \(F\), the models of \(\text{SM}[F; p]\) are the models of \(F\) that are “stable” on intensional predicates \(p\). Formally, \(\text{SM}[F; p]\) is defined as

\[ F \land \neg \exists u((u < p) \land F^*(u)), \]

where \(p\) and \(u\) are defined same as in CIRC\([F; p]\) and \(F^*(u)\) is defined recursively:

- \(p_i(t)^* = u_i(t)\) for any tuple \(t\) of terms;

\[\text{http://www.cs.utexas.edu/users/tag/cmodels.html}\]
\[\text{http://potassco.sourceforge.net/}\]
\[\text{http://www.dbai.tuwien.ac.at/proj/dlv/}\]
• $F^* = F$ for any atomic formula $F$ (including $\bot$ and equality) that does not contain members of $p$;

• $(F \land G)^* = F^* \land G^*$;

• $(F \lor G)^* = F^* \lor G^*$;

• $(F \rightarrow G)^* = (F^* \rightarrow G^*) \land (F \rightarrow G)$;

• $(\forall x F)^* = \forall x F^*$;

• $(\exists x F)^* = \exists x F^*$.

A model of $F$ (in the sense of first-order logic) is $p$-stable if it satisfies SM$[F; p]$. Answer sets are defined as a special class of stable models as follows. Let $\sigma(F)$ be the signature consisting of the object, function and predicate constants occurring in $F$. By $pr(F)$ we denote the list of all predicate constants occurring in $F$. If $F$ contains at least one object constant, an Herbrand interpretation of $\sigma(F)$ that satisfies SM$[F; pr(F)]$ is called an answer set of $F$. The answer sets of a logic program $\Pi$ are defined as the answer sets of the FOL-representation of $\Pi$ (i.e., the conjunction of the universal closure of implications corresponding to the rules). For example, the FOL-representation of the program

\[
p(a) \\
q(b) \\
r(x) \leftarrow p(x), \text{not } q(x)
\]

is

\[
p(a) \land q(b) \land \forall x((p(x) \land \neg q(x)) \rightarrow r(x)).
\]  

It is shown that this definition of answer sets, when applied to the syntax of (disjunctive) logic programs, is equivalent to the traditional definition of answer sets [Gelfond and Lifschitz, 1988] that is based on grounding and fixpoint construction [Ferraris et al., 2007].

4 BMC using the Stable Model Semantics

Due to the expressivity and efficiency of the answer set solvers, as shown in [Heljanko and Niemela, 2001] and [Tang and Ternovska, 2005], implementations of BMC based on ASP is promising. Instead of translating LTL formulas into propositional formulas, the authors translate them into rules in ASP. However, the transformation is not straightforward and uses predicates that are not in the original language. Also, some knowledge of logic programming and the stable model semantics is necessary to understand the translation. An alternative is to turn the LTL formulas to first-order sentences and use the new stable model semantics for first-order sentences. Due to the generality of first-order logic syntax, this translation is more intuitive and results in a more succinct representation of LTL formulas compared to the translation to ASP. In several cases, the alternate representation is also simpler compared to the representation in ASP. For example, consider the simple formula $F = Gp$. In [Tang and Ternovska, 2005], this is represented using the following program:
f(I) :- p(I), f(I+1), has_next_state(I).
f(k) :- le, not q_h.

q_h :- il(I), not p(I), has_next_state(I).
     :- not f(0).

In the above program, the predicates \( f \) and \( q_h \) are the new predicate introduced, \( k \) is the bound, \( le \) represents existence of a loop and \( il(I) \) represents that \( I \) is part of the loop. On the other hand, \( F \) can be represented using in the syntax of first-order logic as

\[
le \land \forall i (i \geq \min(0, \text{loopStart}) \land i \leq k \rightarrow p(i)),
\]

where \( \text{loopStart} \) indicates the starting point of the loop on the path. The figure below gives a brief overview of the differences between different existing approaches for BMC and the proposed approach:

F2LP [Lee and Palla, 2009], which is an implementation of the new stable model semantics can be used for computation. Given any first-order sentence under the stable model semantics, F2LP turns it into a disjunctive logic program, while preserving the answer sets. So, answer set solvers can be used to compute the answer sets of first-order sentences.

5 System F2LP

System F2LP \(^7\) is a step towards implementing the general language of stable models. It translates an arbitrary first-order formula under the stable model semantics into an answer set program. By calling existing answer set solvers on the resulting program, we can compute Herbrand stable models of a first-order formula. The system extends the previous version described in [Lee and Palla, 2007], which computes stable models of arbitrary propositional formulas.

Formulas can be encoded in the language of F2LP using the following ASCII characters.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \neg )</td>
<td>( - )</td>
</tr>
<tr>
<td>( \land )</td>
<td>&amp;</td>
</tr>
<tr>
<td>( \lor )</td>
<td></td>
</tr>
<tr>
<td>( \rightarrow )</td>
<td>( \rightarrow )</td>
</tr>
<tr>
<td>( \perp )</td>
<td>false</td>
</tr>
<tr>
<td>( \top )</td>
<td>true</td>
</tr>
<tr>
<td>( \forall xyz )</td>
<td>![X,Y,Z]:</td>
</tr>
<tr>
<td>( \exists xyz )</td>
<td>?[X,Y,Z]:</td>
</tr>
</tbody>
</table>

F2LP turns a formula into the corresponding LPARSE program.\(^8\) The usual LPARSE encoding is also allowed in F2LP: it is simply copied to the output. The LPARSE program returned

\(^7\) http://reasoning.eas.asu.edu/f2lp.
\(^8\) http://www.tcs.hut.fi/Software/smodels.
by f2lp can be passed to ASP grounders and solvers that accept lparse language. While function symbols are allowed in the input language of f2lp, it is left to the grounder to handle them.

6  Kripke structures to First-Order Formulas

As described in [Biere et al., 1999], a Kripke structure is a tuple $(S, I, T, L)$ where

- $S$ is a finite set of states \( \{s_0, s_1, s_2, \ldots, s_n\} \);
- $I$ is a subset of $S$ defining the initial states;
- $T \subseteq S \times S$ is a transition relation between states; and
- $L$ is a mapping from each state $s$ to a set of atomic propositions that are true in $s$.

We assume from here that $I$ contains only one state. Also, in line with [Biere et al., 1999], we assume that for every state $s$, there is at least one $s'$ such that $(s, s') \in T$.

As an example, we show below the Kripke structure depicting the behaviour of a system controlling the entry of processes into a critical section. In the example, there is one critical section, one semaphore that is used to control the entry into the critical section and 2 processes. $NC0$ indicates that process 0 is not in the critical section, $TRY0$ indicates that process 0 has issued a request for the semaphore and $CS0$ indicates that process 0 is in the critical section.

---

This is taken from the presentation http://iplu.vtt.fi/digitalo/modelchecking.pdf.
\begin{itemize}
  \item $S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8\}$;
  \item $I = \{s_0\};$
  \item $T = \{(s_0, s_0), (s_0, s_1), (s_0, s_2), (s_1, s_3), (s_1, s_4), (s_2, s_5), (s_2, s_6), (s_3, s_7),
  (s_4, s_0), (s_4, s_7), (s_5, s_8), (s_6, s_0), (s_6, s_8), (s_7, s_2), (s_8, s_1)\};$ and
  \item $L(s_0) = \{NC0, NC1\}$, $L(s_1) = \{TRY0, NC1\}$, $L(s_2) = \{NC0, TRY1\}$, $L(s_3) = \{TRY0, TRY1\}$, $L(s_4) = \{CS0, NC1\}$, $L(s_5) = \{TRY0, TRY1, InS5\}$, $L(s_6) = \{NC0, CS1\}$,
  $L(s_7) = \{CS0, TRY1\}$, $L(s_8) = \{TRY0, CS1\}$. \end{itemize}

Note that we use an additional fluent $InS5$ to differentiate between states $s_3$ and $s_5$.

Given a Kripke structure $M=(S, I, T, L)$, the first-order representation of $M$ is shown below. We use $\text{Holds}(f, t)$ to denote that fluent $f$ holds at time $t$, $\text{InState}(s, t)$ to denote that the system is in state $s$ at time $t$ and $\text{Tr}(s, s', t)$ to denote that there is a transition from state $s$ to state $s'$ at time $t$.

(a) Data:
\begin{itemize}
  \item add $Time(0..k)$ where $k$ is a constant representing the bound.
  \item add $\text{State}(s)$ for every state $s \in S$.
  \item add $\text{Tr}(s, s', t)$ for every transition $(s, s') \in T$.
\end{itemize}

(b) State Definitions:
\begin{itemize}
  \item for the initial state $s \in I$, add
  \[
  \bigwedge_{p \in L(s)} \text{Holds}(p, 0) \land \bigwedge_{q \notin L(s)} \neg \text{Holds}(q, 0).
  \]
  \item for every state $s \in S$, add
  \[
  \forall t \left( \bigwedge_{p \in L(s)} \text{Holds}(p, t) \land \bigwedge_{q \notin L(s)} \neg \text{Holds}(q, t) \rightarrow \text{InState}(s, t) \right).
  \]
\end{itemize}

(c) Possible transitions:
\begin{itemize}
  \item for transitions from state $s$, $(s, s_1), (s, s_2), \ldots, (s, s_m)$, add
  \[
  \forall t (\text{InState}(s, t) \land t < k \rightarrow \text{Tr}(s, s_1, t) \lor \ldots \lor \text{Tr}(s, s_m, t)).
  \]
\end{itemize}

(d) Effects:
\begin{itemize}
  \item every transition must change the state appropriately:
  \[
  \forall s, s', t \neg (\text{Tr}(s, s', t) \land \neg \text{InState}(s', t + 1))
  \]
\end{itemize}

(e) To exempt $\text{Holds}$ from minimization, for each fluent $f$, we add:
\[
\forall t (\text{Holds}(f, t) \lor \neg \text{Holds}(f, t)).
\]

We will represent the set of first-order formulas corresponding to a Kripke structure $M$ by $M(k)$. Given below is a partial first-order encoding for the semaphore example.
(a) Data:
\[ \text{Time}(0..k). \]
\[ \text{State}(s_0). \]
\[ T(s_0, s_0) \land T(s_0, s_1) \land T(s_0, s_2). \]
\[ T(s_1, s_3) \land T(s_1, s_4). \]
\[ T(s_2, s_5) \land T(s_2, s_6). \]
\[ T(s_3, s_7). \]
\[ T(s_4, s_0) \land T(s_4, s_7). \]
\[ T(s_5, s_8). \]
\[ T(s_6, s_0) \land T(s_6, s_8). \]
\[ T(s_7, s_2). \]
\[ T(s_8, s_1). \]

(b) State Definition for \( s_0 \):
\[ \forall t \left( \text{Holds}(nc_0, t) \land \text{Holds}(nc_1, t) \land \neg \text{Holds}(try_0, t) \land \neg \text{Holds}(try_1, t) \land \neg \text{Holds}(cs_0, t) \land \neg \text{Holds}(cs_1, t) \land \neg \text{Holds}(ins_5, t) \rightarrow \text{InState}(s_0, t) \right). \]

(c) Possible transitions from \( s_0 \):
\[ \forall t \left( \text{InState}(s_0, t) \land t < k \rightarrow \text{Tr}(s_0, s_0, t) \lor \text{Tr}(s_0, s_1, t) \lor \text{Tr}(s_0, s_2, t) \right). \]

(d) Effects:
\[ \forall s, s', t \left( \neg \left( \text{Tr}(s, s', t) \land \neg \text{InState}(s', t + 1) \right) \right). \]

(e) Choice:
\[ \forall f, t \left( \text{Holds}(f, t) \lor \neg \text{Holds}(f, t) \right). \]

The complete encoding is shown in the appendix.

7 LTL to First-Order Formulas

In line with the approaches in [Biere et al., 1999] and [Heljanko and Niemela, 2001], we will consider two kinds of paths - one with a loop and another without.
∀ss′t(InState(s, k) ∧ T(s, s′) ∧ t ≤ k ∧ InState(s′, t) → Loop(t)) \quad (3)
∃tLoop(t) → LoopExists \quad (4)
∀t(Loop(t) → LoopStart(t) ∨ ¬LoopStart(t)) \quad (5)
¬(LoopExists ∧ ¬∃tLoopStart(t)) \quad (6)
¬∃tt_1(LoopStart(t) ∧ LoopStart(t_1) ∧ t \neq t_1) \quad (7)

The first and second formulas define the conditions under which a loop exists in the path. The remaining formulas define the starting time point of the loop to be considered. If a path has more than one loop, then different answer sets are generated corresponding to the same path and each of these answer sets is generated by considering a different loop in the path. We will represent this set of definitions by Def. We will now show how to recursively turn a LTL formula to a first-order formula. This translation is based on the translation in [Biere et al., 1999]. We will represent the translation by FOL. In the following, we use

- \( t_1 \geq \text{min}(\text{LoopStart}, t) \) as an abbreviation for
  \[ \exists t_2(\text{LoopStart}(t_2) \land t_2 \leq t \land t_1 \geq t_2) \lor \exists t_2(\text{LoopStart}(t_2) \land t < t_2 \land t_1 \geq t) \]

- \( f(\text{LoopStart}, k) \) as an abbreviation for
  \[ \exists t(\text{LoopStart}(t) \land f(t, k)) \]

- \( \text{LoopStart} \leq t \) as an abbreviation for
  \[ \exists t_1(\text{LoopStart}(t_1) \land t_1 \leq t) \]

Translation FOL:

- \( FOL(P(t, k)) \):
  \[ \text{Holds}(p, t) \]

- \( FOL(\neg f(t, k)) \):
  \[ \neg FOL(f(t, k)) \]

- \( FOL(f(t, k) \odot g(t, k)) \):
  \[ FOL(f(t, k)) \odot FOL(g(t, k)) \]
  where \( \odot \in \{\land, \lor, \rightarrow\} \).

- \( FOL(\mathbf{G}f(t, k)) \) where \( \mathbf{G} \) represents the LTL operator \textit{globally}:
  \[ \text{LoopExists} \land \forall t_1(t_1 \geq \text{min}(\text{LoopStart}, t) \land t_1 \leq k \rightarrow FOL(f(t_1, k))) \]
• $FOL(\mathbf{F}f(t,k))$ where $\mathbf{F}$ represents the eventuality operator:
This is turned to the conjunction of the following formulas

$$\text{LoopExists} \rightarrow \exists t_1(t_1 \geq \min(\text{LoopStart}, t) \land t_1 \leq k \land FOL(f(t_1,k))).$$

$$\neg\text{LoopExists} \rightarrow \exists t_1(t_1 \geq t \land t_1 \leq k \land FOL(f(t_1,k))).$$

• $FOL(\mathbf{X}f(t,k))$ where $\mathbf{X}$ represents the operator next time:
This is turned to the conjunction of the following formulas

$$(t < k \rightarrow FOL(f(t + 1, k))).$$

$$(t = k \land \text{LoopExists} \rightarrow FOL(f(\text{LoopStart}, k))).$$

$$\neg(\neg\text{LoopExists} \land t = k).$$

• $FOL((f\mathbf{U}g)(t,k))$ where $\mathbf{U}$ represents the until operator:

$$\exists t_1 \left( t_1 \geq t \land t_1 \leq k \land FOL(g(t_1,k)) \land \forall t_2(t_2 \geq t \land t_2 < t_1 \rightarrow FOL(f(t_2,k))) \right) \lor$$

$$(\text{LoopExists} \land \text{LoopStart} \leq t) \land \exists t_1 \left( t_1 \geq \text{LoopStart} \land t_1 < t \land FOL(g(t_1,k)) \land \forall t_2(t_2 \geq t \land t_2 \leq k \rightarrow FOL(f(t_2,k))) \land \forall t_2(t_2 \geq \text{LoopStart} \land t_2 < t_1 \rightarrow FOL(f(t_2,k))) \right)$$

• $FOL((f\mathbf{R}g)(t,k))$ where $\mathbf{R}$ represents the releases operator:
This will be turned to the conjunction of the following formulas.

$$\neg\text{LoopExists} \rightarrow$$

$$\exists t_1 \left( t_1 \geq t \land t_1 \leq k \land FOL(f(t_1,k)) \land \forall t_2(t_2 \geq t \land t_2 \leq t_1 \rightarrow FOL(g(t_2,k))) \right).$$

$$\text{LoopExists} \rightarrow$$

$$\exists t_1 \left( t_1 \geq t \land t_1 \leq k \land FOL(f(t_1,k)) \land \forall t_2(t_2 \geq t \land t_2 \leq t_1 \rightarrow FOL(g(t_2,k))) \right) \lor$$

$$\forall t_1(t_1 \geq \min(t, \text{LoopStart}) \land t_1 \leq k \rightarrow FOL(g(t_1,k))) \lor$$

$$\exists t_1 \left( t_1 \geq \text{LoopStart} \land t_1 < t \land FOL(f(t_1,k)) \land \forall t_2(t_2 \geq t \land t_2 \leq k \rightarrow FOL(g(t_2,k))) \land \forall t_2(t_2 \geq \text{LoopStart} \land t_2 < t_1 \rightarrow FOL(g(t_2,k))) \right).$$
Given any LTL formula \( F \) and a bound \( k \), the first-order formula representing \( F \) is given by \( FOL(F(0,k)) \). It is easy to check that by grounding \( FOL(F(0,k)) \), we obtain a propositional formula that is synonymous to the propositional representation of \( F \) obtained by using the approach defined in [Biere et al., 1999]. As an example for our translation, consider the LTL formula

\[
F = \text{FG}p.
\]

This is first turned to the conjunction of

\[
\text{LoopExists} \rightarrow \exists t_1 (t_1 \geq \min(\text{LoopStart}, 0) \land t_1 \leq k \land FOL((\text{G}p)(t_1,k))).
\]

\[
\neg\text{LoopExists} \rightarrow \exists t_1 (t_1 \geq 0 \land t_1 \leq k \land FOL((\text{G}p)(t_1,k))).
\]

Expanding the above formulas results in conjunction of

\[
\text{LoopExists} \rightarrow \exists t_1 \left( t_1 \geq \min(\text{LoopStart}, 0) \land t_1 \leq k \land \forall t_2 (t_2 \geq \min(\text{LoopStart}, t_1) \land t_2 \leq k \rightarrow \text{Holds}(p, t_2)) \right)
\]

and

\[
\neg\text{LoopExists} \rightarrow \exists t_1 \left( t_1 \geq 0 \land t_1 \leq k \land \text{LoopExists} \land \forall t_2 (t_2 \geq \min(\text{LoopStart}, t_1) \land t_2 \leq k \rightarrow \text{Holds}(p, t_2)) \right),
\]

which is equivalent to first formula.

The correctness of our approach follows from the following theorem:

**Theorem 1** For any Kripke structure \( M \) and an LTL formula \( G \), there is a path of length \( k \) in \( M \) that satisfies \( G \) iff \( M(k) \land \text{Def} \land \neg\neg FOL(G(0,k)) \) has an answer set.

### 8 Applications and Examples

BMC has already been proven to be very effective in verification of hardware systems. Since the BMC approach exhaustively searches through all the paths in the model, it provides more coverage compared to other verification approaches. However, since the process depends on obtaining the correct model, there are certain drawbacks. The effectiveness of the approach depends on the correctness of the model used to represent the system behaviour. Also, since the model is generally in the form of a Kripke structure, obtaining models for large software systems is more difficult as there can be a very large number of states in such systems. For these reasons, BMC is more effective on software systems where obtaining the correct model is not very complex. Despite these drawbacks, BMC can be quite useful for software verification. For example, since the approach identifies infinite execution using loops in the paths, it can identify situations such as deadlocks. Since on failure the approach produces counter-examples, the approach is very useful for debugging. Consider the semaphore example presented earlier. To check if a requirement \( G \) is satisfied by the model \( M \), we need to encode \( M(k) \land \neg\neg FOL((\text{negation of } G)(0,k)) \) in f2lp. Here we will consider 2 requirements.
• $G = G\neg (cs0 \land cs1)$ - at no time both the processes are in the critical section:

Negation of $G$ is $F(cs0 \land cs1)$. When we encode $M(k) \land \neg\neg FOL(F(cs0 \land cs1)(0,k))$ in f2lp, there are no answer sets returned \(^{11}\). This implies that the model satisfies the requirement. The f2lp encoding is provided in the appendix.

• $G = G(try1 \rightarrow Fcs1)$ - whenever process 1 requests for the semaphore, it eventually gets it:

Negation of $G$ is $F(try1 \land Gcs1)$. When we encode $M(k) \land \neg\neg FOL(F(try1 \land Gcs1)(0,k))$ in f2lp, there are no answer sets returned \(^{12}\). This implies that the model satisfies the requirement. The f2lp encoding is provided in the appendix.

The Kripke structure for the semaphore example satisfies both the requirements mentioned above. However, upon adding $T(s_5,s_7)$ and removing $T(s_5,s_8)$, the resulting structure does not satisfy the second requirement. The following answer set is produced for bound $k=3$:

**Answer:** 1
holds(nc0,0) holds(nc1,0) holds(nc0,1) holds(try1,1) holds(try0,2) holds(try1,2)
holds(ins5,2) holds(try1,3) holds(cs0,3) instate(s0,0) instate(s2,1) instate(s5,2)
instate(s7,3) tr(s5,s7,2) tr(s0,s2,0) tr(s2,s5,1) le

**Models:** 1
**Time:** 0.000 (Parsing: 0.000)

By following the transitions indicated by the predicate tr in the output, we can identify the path that violated the requirement. According to the above output, the path $s_0 \rightarrow s_2 \rightarrow s_5 \rightarrow s_7$ violated the requirement. This can be easily verified by observing that the path contains the loop $s_2 \rightarrow s_5 \rightarrow s_7$ and in each of the states in the loop $cs1$ is false.

We also applied the proposed method to the elevator example, which is shown in the following state diagram.

\(^{11}\)This was verified for upto bound $k=20$.

\(^{12}\)This was verified for upto bound $k=20$. 

---

13
We checked the system for the requirements:

- $G\neg(up \wedge down)$

- $G(on \wedge open \wedge f3 \wedge up \rightarrow F(on \wedge close \wedge f1 \wedge down))$ - if the elevator is in state $\{on, open, f3, up\}$, then eventually it will reach the state $\{on, close, f1, down\}$.

The first requirement is satisfied whereas the second requirement is not. The below answer set depicts the path that violates the second requirement.

\textbf{Answer: 1}
holds(off,0) holds(on,1) holds(close,1) holds(f1,1)
holds(down,1) holds(on,2) holds(close,2) holds(f2,2) holds(up,2)
holds(on,3) holds(close,3) holds(f3,3) holds(up,3) holds(on,4)
holds(open,4) holds(f3,4) holds(up,4)
instate(s0,0) instate(s2,1)
instate(s5,2) instate(s7,4) instate(s8,3) tr(s0,s2,0) tr(s8,s7,3)
tr(s5,s8,2) tr(s2,s5,1) le

Models : 1
Time : 0.000 (Parsing: 0.000)

The corresponding f2lp input can be found in the appendix section.

9 Comparison with existing approaches

We compare our approach to the SAT-based approach in [Biere et al., 1999] and the ASP-based approaches in [Heljanko and Niemela, 2001] and [Tang and Ternovska, 2005]. Since the SAT based approach in [Biere et al., 1999] is based on propositional logic, the SAT encoding needs to be updated every time the bound is changed. On the other hand, since our approach can use variables, the encoding of the state transitions and the LTL formulas remains the same even when the bound is updated. In order to verify with a new bound, the bound \((k)\) just needs to be updated at run-time. The approach in [Heljanko and Niemela, 2001] uses Petri nets and the approach in [Tang and Ternovska, 2005] considers the problem of BMC for abstract state machines. Though those approaches can use variables, the ASP representation of LTL formulas is not straightforward and some expertise in ASP is required to understand them. Also, for several LTL formulas, the first-order representation is simpler compared to the ASP representation provided in those papers. One such example was presented earlier in section 4.

10 Conclusion and Future Work

In this report, we presented a short survey of model checking approaches and proposed a new approach for BMC using the general language of the stable model semantics. We also explained the advantages of our approach when compared to the existing SAT-based and ASP-based approaches. Future work includes extending our approach to deal with temporal logics with explicit time which we think are more suitable for applications where activities generally have certain time constraints. Another direction in our future work would be to further simply the first-order representations of LTL formulas involving operators \(U\) and \(R\), probably by using function constants.
11 Proofs

11.1 Proof of Theorem 1

We will represent the set of state definitions as $St$, the set of formulas defining the possible transitions as $PT$ and the formula defining the effects of transitions as $Eff$. Additionally, we will use $Data$ to represent the data. For our purpose, it is sufficient to consider the facts about the transitions $T(s, s')$ as the other facts just define the domains for time and state.

Answer sets of $M(k) \land Def \land \neg\neg FOL(G(0, k))$ are the Herbrand models of

$$SM[Data \land St \land PT \land Eff \land Def \land \neg\neg FOL(G(0, k)); pr(M(k)) \setminus \{Holds\}].$$

From the splitting theorem in [Ferraris et al., 2009], it follows that the above formula is equivalent to

$$SM[Data \land St \land PT \land Eff; T, InState, Tr] \land SM[Def \land \neg\neg FOL(G(0, k)); LoopExists, Loop, LoopStart],$$

which is in turn equivalent to

$$SM[Data; T] \land SM[St; InState] \land SM[PT; Tr] \land Eff \land$$

$$SM[(3); Loop] \land SM[(4); LoopExists] \land SM[(5); LoopStart] \land (6) \land (7) \land \neg\neg FOL(G(0, k)).$$

From the completion lemma in [Ferraris et al., 2010], it follows that the above formula is equivalent to \(^{13}\)

$$comp_T(Data) \land comp_{InState}(St) \land SM[PT; Tr] \land Eff \land$$

$$comp_{Loop}(3) \land comp_{LoopExists}(4) \land SM[(5); LoopStart] \land (6) \land (7) \land \neg\neg FOL(G(0, k)).$$

Applying shifting, we get that the above formula is equivalent to

$$comp_T(Data) \land comp_{InState}(St) \land shift(PT) \land Eff \land$$

$$comp_{Loop}(3) \land comp_{LoopExists}(4) \land (5') \land (6) \land (7) \land \neg\neg FOL(G(0, k)).$$

where $shift(PT)$ is defined as the set of formulas

$$InState(s, t) \land \bigwedge_{i \neq j} \neg Tr(s, s_i, t) \leftrightarrow Tr(s, s_j, t)$$

for every

$$InState(s, t) \leftrightarrow \bigvee_i Tr(s, s_i, t)$$

in $PT$, and $(5')$ stands for the formula

$$\forall t (Loop(t) \land \neg\neg LoopStart(t) \leftrightarrow LoopStart(t)).$$

\(^{13}\)By $comp_P(F)$, we denote the completion of $F$ with respect to the predicate $P$.
Now, consider the formula

\[ \text{comp}_T(\text{Data}) \land \text{comp}_{\text{InState}}(\text{St}) \land \text{shift}(\text{PT}) \land \text{Eff}. \]

Since we are considering only the Herbrand models, we can identify interpretations with ground atoms. Assume that initial state is \( s_0 \). For any model \( X \), it is clear that \( \text{InState}(s_0, 0) \in X \) holds. From \( \text{shift}(\text{PT}) \), it follows that there is exactly one \( \text{Tr}(s_0, s_i, 0) \) such that \( \text{Tr}(s_0, s_i, 0) \in X \). Let that \( s_i \) be \( s_1 \) without loss of generality. From \( \text{Eff} \), it follows that \( \text{InState}(s_1, 1) \in X \) and \( \text{Holds}(p, 1) \in X \) iff \( p \) is true in \( s_1 \). Again from \( \text{shift}(\text{PT}) \), it follows that there is exactly one \( \text{Tr}(s_1, s_j, 1) \) such that \( \text{Tr}(s_1, s_j, 1) \in X \). Let that \( s_j \) be \( s_2 \). Continuing this way, we get a sequence of states \( s_0, s_1, s_2, \ldots, s_k \) such that for every \( s_i, s_{i+1}, \text{Tr}(s_i, s_{i+1}, i) \in X \), \( \text{InState}(s_i, i) \in X \) and \( \text{Holds}(p, i) \in X \) iff \( p \) holds in \( s_i \). So, every Herbrand model that satisfies \( \text{comp}_T(\text{Data}) \land \text{comp}_{\text{InState}}(\text{St}) \land \text{shift}(\text{PT}) \land \text{Eff} \) defines exactly one path of length \( k \) in \( M \). Also, for every path of length \( k \) in \( M \) there is a Herbrand model of \( \text{comp}_T(\text{Data}) \land \text{comp}_{\text{InState}}(\text{St}) \land \text{shift}(\text{PT}) \land \text{Eff} \) because of the formula \( \text{shift}(\text{PT}) \). In the following, \( X \) represents a Herbrand model of \( \text{comp}_T(\text{Data})\land\text{comp}_{\text{InState}}(\text{St})\land\text{shift}(\text{PT})\land\text{Eff} \) and \( Y \) represents a Herbrand model of

\[ \text{comp}_T(\text{Data}) \land \text{comp}_{\text{InState}}(\text{St}) \land \text{shift}(\text{PT}) \land \text{Eff} \land \text{comp}_E(3) \land \text{comp}_{\text{LoopExists}}(4) \land (5') \land (6) \land (7). \]

We say \( Y \) is an extension of \( X \) if \( X \) can be obtained from \( Y \) by removing all the atoms containing predicates from \( \{\text{LoopExists}, \text{LoopStart}, \text{Loop}\} \). It is clear that for any \( Y \), there is an \( X \) such that \( Y \) is an extension of \( X \). Also, for any \( X \), there is a \( Y \) such that \( Y \) is an extension of \( X \). In view of \( (5') \land (6) \land (7) \), we can conclude that

- every \( X \) defines exactly one path of length \( k \) in \( M \) and for every path of length \( k \) in \( M \), there is an \( X \) that defines it.
- if the path defined by \( X \) does not contain a loop, then there is exactly one extended model \( Y \) of \( X \), and \( Y \approx X \). For any \( Y \) such that \( \text{LoopExists} \notin Y \), there is an \( X \) such that \( X \) defines a loop-free path and \( Y \) is an extension of \( X \).
- if the path defined by \( X \) contains \( n \) loops starting at \( l_1, \ldots, l_n \), then there are exactly \( n \) extended models \( Y_1, \ldots, Y_n \) of \( X \) such that \( \text{LoopStart}(l_i) \in Y_i \) and each \( Y_i \) contains exactly one \( \text{LoopStart}(l_j) \). For any \( Y \) such that \( \text{LoopExists} \in Y \), there is an \( X \) such that \( X \) defines a path with a loop and \( Y \) is an extension of \( X \).

This implies that there is a Herbrand model \( Z \) of

\[ \text{comp}_T(\text{Data}) \land \text{comp}_{\text{InState}}(\text{St}) \land \text{shift}(\text{PT}) \land \text{Eff} \land \text{comp}_E(3) \land \text{comp}_{\text{LoopExists}}(4) \land (5') \land (6) \land (7) \land \neg \neg \text{FOL}(G(0, k)) \]

such that \( \text{LoopExists} \notin Z \) iff there is a loop-free path of length \( k \) that satisfies \( \neg \neg \text{FOL}(G(0, k)) \), and there is a Herbrand model \( Z \) of

\[ \text{comp}_T(\text{Data}) \land \text{comp}_{\text{InState}}(\text{St}) \land \text{shift}(\text{PT}) \land \text{Eff} \land \neg \neg \text{FOL}(G(0, k)). \]
\[ \text{comp}_{\text{Loop}}(3) \land \text{comp}_{\text{LoopExists}}(4) \land (5') \land (6) \land (7) \land \neg\neg\text{FOL}(G(0, k)) \]

with \( \text{LoopExists} \in Z \) iff there is a path of length \( k \) with a loop that satisfies \( \neg\neg\text{FOL}(G(0, k)) \).
Hence the result. ■

References


12 Appendix

F2LP encoding of the Kripke structure for the semaphore example:

time(0..k).
#domain time(I;I1;I2).
state(s0;s1;s2;s3;s4;s5;s6;s7;s8).
#domain state(S;SA;SB;SC;SD).
fluent(nc0;nc1;try0;try1;cs0;cs1;ins5).
#domain fluent(F).

t(s0,s0).t(s0,s1).t(s0,s2).
t(s1,s4).t(s1,s3).
t(s2,s5).t(s2,s6).
t(s3,s7).
t(s4,s7).t(s4,0).
t(s5,s8).
t(s6,s8).t(s6,s0).
t(s7,s2).
t(s8,s1).

holds(nc0, 0) & holds(nc1, 0) & - holds(try0, 0) &
-holds(try1, 0) & -holds(cs0, 0) & -holds(cs1, 0) & -holds(ins5,0).

holds(nc0, I) & holds(nc1, I) & - holds(try0, I) &
-holds(try1, I) & -holds(cs0, I) & -holds(cs1, I) & -holds(ins5,I)
  -> instate(s0,I).

-holds(nc0, I) & holds(nc1, I) & holds(try0, I) &
-holds(try1, I) & -holds(cs0, I) & -holds(cs1, I) & -holds(ins5,I)
  -> instate(s1,I).

holds(nc0, I) & -holds(nc1, I) & holds(try0, I) &
holds(try1, I) & -holds(cs0, I) & -holds(cs1, I) & -holds(ins5,I)
  -> instate(s2,I).

-holds(nc0, I) & -holds(nc1, I) & holds(try0, I) &
holds(try1, I) & -holds(cs0, I) & -holds(cs1, I) & -holds(ins5,I)
  -> instate(s3,I).
-holds(nc0, I) & holds(nc1, I) & -holds(try0, I) &
  -holds(try1, I) & holds(cs0, I) & -holds(cs1, I) & -holds(ins5,I)
  -> instate(s4,I).

-holds(nc0, I) & -holds(nc1, I) & holds(try0, I) &
  holds(try1, I) & -holds(cs0, I) & -holds(cs1, I) & holds(ins5,I)
  -> instate(s5,I).

-holds(nc0, I) & -holds(nc1, I) & -holds(try0, I) &
  -holds(try1, I) & -holds(cs0, I) & holds(cs1, I) & -holds(ins5,I)
  -> instate(s6,I).

-holds(nc0, I) & -holds(nc1, I) & -holds(try0, I) &
  -holds(try1, I) & holds(cs0, I) & -holds(cs1, I) & -holds(ins5,I)
  -> instate(s7,I).

-holds(nc0, I) & -holds(nc1, I) & holds(try0, I) &
  -holds(try1, I) & -holds(cs0, I) & holds(cs1, I) & -holds(ins5,I)
  -> instate(s8,I).

% s0 to s0,s1,s2
instate(s0,I) & I < k -> tr(s0,s0,I) | tr(s0,s1,I) | tr(s0,s2,I).

% s1 to s3,s4
instate(s1,I) & I < k -> tr(s1,s3,I) | tr(s1,s4,I).

% s2 to s5,s6
instate(s2,I) & I < k -> tr(s2,s5,I) | tr(s2,s6,I).

% s3 to s7
instate(s3,I) & I < k -> tr(s3,s7,I).

% s4 to s7,s0
instate(s4,I) & I < k -> tr(s4,s7,I) | tr(s4,s0,I).

% s5 to s8
instate(s5,I) & I < k -> tr(s5,s8,I).

% s6 to s8,s0
instate(s6,I) & I < k -> tr(s6,s8,I) | tr(s6,s0,I).

% s7 to s2
% s7 to s2
instate(s7,I) & I < k -> tr(s7,s2,I).

% s8 to s1
instate(s8,I) & I < k -> tr(s8,s1,I).

% effects
-(tr(SA,SB,I) & -instate(SB,I+1)).

% choice rule for holds.
{holds(F,I)}.

f2lp encoding of the standard definitions

% standard definitions
% existence of a loop
instate(S,k) & t(S,SA) & instate(SA,I) -> loop(I).
loop(I) -> le.

% starting point of the loop
loop(I) -> loopstart(I) | -loopstart(I).
-(le & -?[I]:loopstart(I)).
-?[I,I1]: (loopstart(I) & loopstart(I1) & I != I1).

f2lp encodings of the negation of the requirements:

- F(cs0 & cs1) (negation of G¬(cs0 & cs1)):
  --?[I]: (I <= k & holds(cs0,I) & holds(cs1,I)).

- F(try1 & G¬cs1) (negation of G(try1 → Fcs1)):
  --(le & ?[I]: ( holds(try1,I) & ![I1]: (?[I2]: (loopstart(I2) & I2 >= I & I1 >= I) | ?[I2]: (loopstart(I2) & I2 < I & I1 >= I2) -> -holds(cs1,I1) ) )).

f2lp encoding of the modified Kripke structure for the semaphore example (modified by adding a transition from s5 to s7 and deleting the transition from s5 to s8):

time(0..k).
#domain time(I;I1;I2).
state(s0;s1;s2;s3;s4;s5;s6;s7;s8).
#domain state(S;SA;SB;SC;SD).
fluent(nc0;nc1;try0;try1;cs0;cs1;ins5).
#domain fluent(F).

t(s0,s0).t(s0,s1).t(s0,s2).
t(s1,s4).t(s1,s3).
t(s2,s5).t(s2,s6).
t(s3,s7).
t(s4,s7).t(s4,0).

%introducing an error so that there is a chance that
%process 1 does not get the semaphore after it tries to get it

t(s6,s8).t(s6,s0).
t(s7,s2).
t(s8,s1).

holds(nc0, 0) & holds(nc1, 0) & - holds(try0, 0) &
    -holds(try1, 0) & -holds(cs0, 0) & -holds(cs1, 0) & -holds(ins5,0).

holds(nc0, I) & holds(nc1, I) & - holds(try0, I) &
    -holds(try1, I) & -holds(cs0, I) & -holds(cs1, I) & -holds(ins5,I)
    -> instate(s0,I).

-holds(nc0, I) & holds(nc1, I) & holds(try0, I) &
    -holds(try1, I) & -holds(cs0, I) & -holds(cs1, I) & -holds(ins5,I)
    -> instate(s1,I).

holds(nc0, I) & -holds(nc1, I) & -holds(try0, I) &
    holds(try1, I) & -holds(cs0, I) & -holds(cs1, I) & -holds(ins5,I)
    -> instate(s2,I).

-holds(nc0, I) & -holds(nc1, I) & holds(try0, I) &
    holds(try1, I) & -holds(cs0, I) & -holds(cs1, I) & -holds(ins5,I)
    -> instate(s3,I).

-holds(nc0, I) & holds(nc1, I) & -holds(try0, I) &
    -holds(try1, I) & holds(cs0, I) & -holds(cs1, I) & -holds(ins5,I)
    -> instate(s4,I).

-holds(nc0, I) & -holds(nc1, I) & holds(try0, I) &
    holds(try1, I) & -holds(cs0, I) & -holds(cs1, I) & holds(ins5,I)
    -> instate(s5,I).

holds(nc0, I) & -holds(nc1, I) & -holds(try0, I) &
\(-\text{holds}(\text{try}1, I) \& \-\text{holds}(\text{cs}0, I) \& \text{holds}(\text{cs}1, I) \& \-\text{holds}(\text{ins}5,I)\)
\rightarrow \text{instate}(\text{s}6,I).

\(-\text{holds}(\text{nc}0, I) \& \-\text{holds}(\text{nc}1, I) \& \-\text{holds}(\text{try}0, I) \&
\text{holds}(\text{try}1, I) \& \text{holds}(\text{cs}0, I) \& \-\text{holds}(\text{cs}1, I) \& \-\text{holds}(\text{ins}5,I)\)
\rightarrow \text{instate}(\text{s}7,I).

\(-\text{holds}(\text{nc}0, I) \& \-\text{holds}(\text{nc}1, I) \& \text{holds}(\text{try}0, I) \&
\-\text{holds}(\text{try}1, I) \& \-\text{holds}(\text{cs}0, I) \& \text{holds}(\text{cs}1, I) \& \-\text{holds}(\text{ins}5,I)\)
\rightarrow \text{instate}(\text{s}8,I).

\%
\text{s}0 \text{ to } \text{s}0,\text{s}1,\text{s}2
\text{instate}(\text{s}0,I) \& I < k \rightarrow \text{tr}(\text{s}0,\text{s}0,I) \mid \text{tr}(\text{s}0,\text{s}1,I) \mid \text{tr}(\text{s}0,\text{s}2,I).

\%
\text{s}1 \text{ to } \text{s}3,\text{s}4
\text{instate}(\text{s}1,I) \& I < k \rightarrow \text{tr}(\text{s}1,\text{s}3,I) \mid \text{tr}(\text{s}1,\text{s}4,I).

\%
\text{s}2 \text{ to } \text{s}5,\text{s}6
\text{instate}(\text{s}2,I) \& I < k \rightarrow \text{tr}(\text{s}2,\text{s}5,I) \mid \text{tr}(\text{s}2,\text{s}6,I).

\%
\text{s}3 \text{ to } \text{s}7
\text{instate}(\text{s}3,I) \& I < k \rightarrow \text{tr}(\text{s}3,\text{s}7,I).

\%
\text{s}4 \text{ to } \text{s}7,\text{s}0
\text{instate}(\text{s}4,I) \& I < k \rightarrow \text{tr}(\text{s}4,\text{s}7,I) \mid \text{tr}(\text{s}4,\text{s}0,I).

\%
\text{introducing an error so that there is a chance that}
\%	ext{process 1 does not get the semaphore after it tries to get it}
\%
\text{s}5 \text{ to } \text{s}7
\text{instate}(\text{s}5,I) \& I < k \rightarrow \text{tr}(\text{s}5,\text{s}7,I).

\%
\text{s}6 \text{ to } \text{s}8,\text{s}0
\text{instate}(\text{s}6,I) \& I < k \rightarrow \text{tr}(\text{s}6,\text{s}8,I) \mid \text{tr}(\text{s}6,\text{s}0,I).

\%
\text{s}7 \text{ to } \text{s}2
\text{instate}(\text{s}7,I) \& I < k \rightarrow \text{tr}(\text{s}7,\text{s}2,I).

\%
\text{s}8 \text{ to } \text{s}1
\text{instate}(\text{s}8,I) \& I < k \rightarrow \text{tr}(\text{s}8,\text{s}1,I).

\%
\text{effects}
\-(\text{tr}(\text{SA},\text{SB},I) \& \-\text{instate}(\text{SB},I+1)).
%choice rule for holds.
{holds(F,I)}.

F2LP encoding of the Kripke structure for the elevator example:

time(0..k).
#domain time(I;I1;I2).
state(s0;s1;s2;s3;s4;s5;s6;s7;s8).
#domain state(S;SA;SB;SC;SD).
fluent(on;off;open;close;f1;f2;f3;up;down).
#domain fluent(F).

t(s0,s0). t(s0,s2). t(s1,s2). t(s2,s0). t(s2,s1). t(s2,s5). t(s3,s5).
t(s4,s6). t(s5,s2). t(s5,s3). t(s5,s8). t(s6,s2). t(s6,s4). t(s6,s8).
t(s7,s8). t(s8,s6). t(s8,s7).

holds(off, 0) & -holds(on, 0) & -holds(open, 0) &
-holds(close, 0) & -holds(f1, 0) & -holds(f2, 0) & -holds(f3,0) &
-holds(up,0) & -holds(down,0).

holds(off, I) & -holds(on, I) & -holds(open, I) &
-holds(close, I) & -holds(f1, I) & -holds(f2, I) & -holds(f3,I) &
-holds(up,I) & -holds(down,I)
-> instate(s0,I).

-holds(off, I) & holds(on, I) & holds(open, I) &
-holds(close, I) & holds(f1, I) & -holds(f2, I) & -holds(f3,I) &
-holds(up,I) & holds(down,I)
-> instate(s1,I).

-holds(off, I) & holds(on, I) & -holds(open, I) &
holds(close, I) & holds(f1, I) & -holds(f2, I) & -holds(f3,I) &
-holds(up,I) & holds(down,I)
-> instate(s2,I).

-holds(off, I) & holds(on, I) & holds(open, I) &
-holds(close, I) & -holds(f1, I) & holds(f2, I) & -holds(f3,I) &
holds(up,I) & -holds(down,I)
-> instate(s3,I).

-holds(off, I) & holds(on, I) & holds(open, I) &
-holds(close, I) & -holds(f1, I) & holds(f2, I) & -holds(f3,I) &
-holds(up,I) & holds(down,I)
-> instate(s4,I).

-holds(off, I) & holds(on, I) & -holds(open, I) &
  holds(close, I) & -holds(f1, I) & holds(f2, I) & -holds(f3,I) &
  holds(up,I) & -holds(down,I)
  -> instate(s5,I).

-holds(off, I) & holds(on, I) & -holds(open, I) &
  holds(close, I) & -holds(f1, I) & holds(f2, I) & holds(f3,I) &
  -holds(up,I) & holds(down,I)
  -> instate(s6,I).

-holds(off, I) & holds(on, I) & holds(open, I) &
  -holds(close, I) & -holds(f1, I) & -holds(f2, I) & holds(f3,I) &
  holds(up,I) & -holds(down,I)
  -> instate(s7,I).

-holds(off, I) & holds(on, I) & -holds(open, I) &
  holds(close, I) & -holds(f1, I) & -holds(f2, I) & holds(f3,I) &
  holds(up,I) & holds(down,I)
  -> instate(s8,I).

instate(s0,I) & I < k -> tr(s0,s0,I) | tr(s0,s2,I).
instate(s1,I) & I < k -> tr(s1,s2,I).
instate(s2,I) & I < k -> tr(s2,s0,I) | tr(s2,s1,I) | tr(s2,s5,I).
instate(s3,I) & I < k -> tr(s3,s5,I).
instate(s4,I) & I < k -> tr(s4,s6,I).
instate(s5,I) & I < k -> tr(s5,s2,I) | tr(s5,s3,I) | tr(s5,s8,I).
instate(s6,I) & I < k -> tr(s6,s2,I) | tr(s6,s4,I) | tr(s6,s8,I).
instate(s7,I) & I < k -> tr(s7,s8,I).
instate(s8,I) & I < k -> tr(s8,s6,I) | tr(s8,s7,I).

%effects
-(tr(SA,SB,I) & -instate(SB,I+1)).

%choice rule for holds.
{holds(F,I)}.

% Encoding of F (up & down).
--?[I]:((I <= k & holds(up,I) & holds(down,I)).

% Encoding of F (on & open & f3 & up & G (-on | -close | -f1 | -down)).
--(le & ?[I]: ( holds(on,I) & holds(open,I) & holds(f3,I) & holds(up,I) &

 ![I1]: (?[I2]: (loopstart(I2) & I2 >= I & I1 >= I) |

 ?[I2]: (loopstart(I2) & I2 < I & I1 >= I2) ->

 -holds(on,I1) | -holds(close,I1) | -holds(f1,I1) | -holds(down,I1)))).