社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
CST2009-4 (2009-6)

SAT and SMT Based Model Checking of Concurrent Systems

Tatsuhiro TSUCHIYA† and Tohru KIKUNO†

† Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita, Osaka 565-0851
E-mail: †{t-tutiya,kikuno}@ist.osaka-u.ac.jp

Abstract We discuss model checking that uses a SAT (satisfiability) or SMT (satisfiability modulo theory) solver. The basic idea behind this model checking approach is to reduce the model checking problem to the satisfiability problem of a formula of some logic. Recent advances in SAT and SMT solvers make this particular approach significantly attractive. However, it does not work effectively in verification of concurrent systems, because the size of the formula blows up if the system has high concurrency. To overcome this challenge, we propose a new semantics for concurrent systems. The new semantics allows a compact formula representation of the behavior of concurrent systems. In this paper, we first introduce this new semantics and bounded model checking based on it, in the context of a general model of concurrent systems. Then we apply it to two specific concurrent system models, namely Petri nets and concurrent programs using unbounded integer variables.

Key words Model checking, concurrent systems, SAT, SMT

1. Introduction

Model checking is a popular formal verification technique. In this paper we discuss the application of model checking to concurrent systems. Specifically we consider model checking that uses a SAT (satisfiability) or SMT (satisfiability modulo theory) solver. The basic idea behind this type of model checking is to reduce the model checking problem to the satisfiability problem of a formula of some logic. In a decade, algorithms for SAT and SMT solving have been significantly improved. These recent advances make this particular model checking approach very attractive.

In verification of concurrent systems, however, SAT or SMT-based model checking does not work effectively. The reason for this is that the size of the formula to be checked blows up if the system has high concurrency.

The purpose of our work is to make SAT and SMT-based model checking a practically feasible approach to verification of concurrent systems. To this end, we propose a new semantic of the behavior of concurrent systems. The new semantics allows one to very compactly represent the behavior of concurrent systems.

In the following sections, we first introduce this new semantics in the context of a general model of concurrent systems. Then we discuss the application of it to two particular instances of concurrent systems: Petri nets and concurrent systems with unbounded integer variables.

2. Preliminaries

2.1 Concurrent Systems

We use a TLA [3]-like logic to specify concurrent systems. A *predicate* is a boolean-valued formula over variables. We denote as $val[P]$ the value of a predicate P when val is the value assignment to the variables occurring in P.

We define a concurrent system as a triple $C \triangleq \langle V, I, A \rangle$, where

- V is a finite set of system variables.
- I is the initial condition of the system.
- A is a finite set of actions.

A *state* is a values assignment to the variables in V. I is a predicate over V such that $s[I] = \text{TRUE}$ iff s is an initial state of the system. An action $a \in A$ is a predicate over $V \cup V'$, where V' is a set of next state variables. For brevity, we let $s[a]s' \triangleq \langle s, s' \rangle [a]$, where s and s' are a state and a next state.

We denote by ENABLED a the condition that the action a is enabled. That is, ENABLED $a = \text{TRUE}$ for state s iff there is some state s' such that $s[a]s' = \text{TRUE}$.

The *transition relation* T is a predicate over V and V' that holds for and only for a state pair $\langle s, s' \rangle$ iff (i) there is some action $a \in A$ such that $s[a]s' = \text{TRUE}$ or (ii) there is no such action and $s = s'$. We have:

$$T \triangleq \bigvee_{a \in A} a \vee \Big(\bigwedge_{a \in A} \text{ENABLED } a = \text{FALSE} \wedge \bigwedge_{v \in V} v' = v \Big)$$

A *run* is an infinite sequence of states $s_0, s_1, s_2, \cdots$ such that:

- I holds for s_1; that is, $s_1[I] = \text{TRUE}$.
- $s_i[T]s_{i+1} = \text{TRUE}$ for any $\langle s_i, s_{i+1} \rangle, i \geqq 0$.

2.2 Ordinary Bounded Model Checking of Safety Properties

For simplicity, we limit our discussion to the verification of reachability of the states where a predicate P holds. In the ordinary bounded model checking, this is done by checking the satisfiability of the following formula Φ_k:

$$\begin{aligned} &Pre_k \triangleq I(V_0/V) \wedge T(V_0/V, V_1/V') \wedge \cdots \wedge T(V_{k-1}/V, V_k/V') \\ &P_k \triangleq P(V_0/V) \vee \cdots \vee P(V_k/V) \\ &\Phi_k \triangleq Pre_k \wedge P_k \end{aligned}$$

where V_i is a set of variables representing the state after i steps and (V_i/V) means replacing variables in V with those in V_i.

Clearly, $\langle s_0, s_1, \cdots, s_k \rangle [Pre_k] = \text{TRUE}$ iff $s_0, s_1, \cdots, s_k$ is a prefix of a run. Hence Φ_k is satisfiable iff some state where p holds is reachable from one of the initial states in at most k steps.

3. Proposed Approach

The problem with using the above formula is that its size blows up when a large number of concurrent actions are possible. Our approach overcomes this problem by introducing a new semantics of concurrent behaviors.

Let $O : A \longrightarrow \{1, ..., n\}$ be a one-to-one map that maps an action to an integer ranging between 1 and n. We denote by a_i the action a such that $O(a) = i$.

In our proposed semantics, a *run* is defined as an infinite sequence of states $s_0, s_1, \cdots$ such that:

- I holds for s_0.
- For $\langle s_j, s_{j+1} \rangle, j \geqq 0$, either $s_j[a_i]s_{j+1} = \text{TRUE}$ or $s_j = s_{j+1}$, or both, where $i = (j \bmod n) + 1$.

It is straightforward to show that for the two semantics, a run in one semantics always has a *stuttering equivalent* run in the other. It has been shown that any LTL$_{-X}$ property is invariant under stuttering [8]. Reachability is a LTL$_{-X}$ property (written as $\Box P$); thus reachability holds in the ordinary semantics iff it does so in the new semantics.

For an action $a \in A$, we define:

$$\hat{a} \triangleq a \vee \bigwedge_{v \in V} v' = v$$

It is easy to see that $s[\hat{a}]s' = \text{TRUE}$ iff $s[a]s'$ or $s = s'$. Hence the prefix of a run of length $n * k$ can be represented as:

$$\begin{aligned} &\hat{Pre}_k \triangleq \\ &I(V_0/V) \\ &\wedge a_1(V_0/V, V_1/V') \wedge \cdots \wedge a_n(V_{n-1}/V, V_n/V') \\ &\wedge a_1(V_n/V, V_{n+1}/V') \wedge \cdots \wedge a_n(V_{2n-1}/V, V_{2n}/V') \\ &\cdots \\ &\wedge a_1(V_{(k-1)*n}/V, V_{(k-1)*n+1}/V') \wedge \\ &\qquad \cdots \wedge a_n(V_{k*n-1}/V, V_{k*n}/V') \end{aligned}$$

Note that in the new semantics, if $s_0 s_1 \cdots s_i s_{i+1} \cdots$ is a run, then $s_0 s_1 \cdots s_i s_i s_i \cdots$ is also a run. Hence, if a state can be reached in exactly i steps, then for any $j \geqq i$, a run always exists where that state is reached in exactly j steps. Consequently, bounded reachability checking can be performed by checking the satisfiability of the formula $\hat{\Phi}_k$:

$$\hat{\Phi}_k \triangleq \hat{Pre}_k \wedge P(V_{k*n}/V)$$

The crucial property of our semantics is that it allows $\hat{\Phi}_k$ to be transformed into a very compact representation. Let V^i be the set of variables that never change their value in action a_i. Thus $\hat{a}_i$ can be written in the following form: Then we have:

$$a_i \equiv a_i \wedge \bigwedge_{v \in V \backslash V^i} v' = v$$

where a_i is $\hat{a}_i$ with $v' = v$ removed for all $v \in V \backslash V^i$. In turn we have:

$$\hat{a}_i \equiv (a_i \vee \bigwedge_{v \in V^i} v' = v) \wedge \bigwedge_{v \in V \backslash V^i} v' = v$$

For each $\hat{a}_i(V_j/V, V_{j+1}/V')$ in $\hat{\Phi}_k$, term $v_{j+1} = v_j$ can be removed for all $v \in V \backslash V^i$, since v_{j+1} can be quantified away as follows: Note that $\hat{\Phi}_k$ can be written as $\hat{\Phi}_k \equiv \hat{\Phi}_k^* \wedge v_{j+1} = v_j$. Because the term $v_{j+1} = v_j$ occurs as a conjunct, $\hat{\Phi}_k$ evaluates to true only if v_{j+1} and v_j have the same value. Hence $\hat{\Phi}_k^*$ with v_{j+1} being replaced with v_j has the same satisfiability as $\hat{\Phi}_k$. The other terms remaining in $\hat{\Phi}_k^*$ can also be removed in the same way.

4. Petri Nets

Syntactically, a *Petri net* is a 4-tuple $(\mathcal{P}, \mathcal{T}, \mathcal{F}, M_0)$ where $\mathcal{P} = \{p_1, p_2, \cdots, p_m\}$ is a finite set of places, $\mathcal{T} = \{t_1, t_2, \cdots, t_n\}$ $(\mathcal{P} \cap \mathcal{T} = \emptyset)$ is a finite set of transitions, $\mathcal{F} \subseteqq (\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P})$ is a set of arcs, and M_0 is the initial state (marking). The set of input places and the set of output places of t are denoted by ${}^\bullet t$ and $t^\bullet$, respectively.

A Petri net is said to be *1-bounded* or *safe* if the number of tokens in each place does not exceed one for any state reachable from the initial state. Note that no source transition (a transition without any input place) exists if a Petri net is safe. For example, the Petri net shown in Figure 1 is safe.

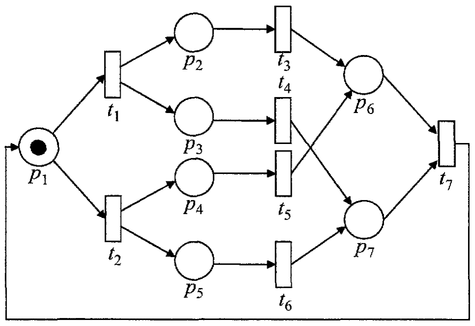

Figure 1 A safe Petri net

Here we show how our proposed approach can apply to a safe Petri net. To this end we first show that a safe Petri net can be described by the general model (V, I, A) presented in Section 2.

First we let:

$$V \triangleq \{v_1, \cdots, v_m\}$$

where v_i is a boolean variable that takes TRUE iff place p_i has a token. Any marking can be uniquely represented by a valuation of these n variables.

M_0 is the only initial state; thus we have:

$$I \triangleq \bigwedge_{p_i \in P_0} v_i = \text{TRUE} \wedge \bigwedge_{p_i \in \mathcal{P} \setminus P_0} v_i = \text{FALSE}$$

where P_0 is the set of places marked with a token in M_0.

Each action corresponds to a transition in $\mathcal{T}$. Let a_t be the action representing transition t. We have:

$$\begin{aligned} a_t \triangleq & \\ & \bigwedge_{p_i \in {}^\bullet t} v_i = \text{TRUE} \wedge \bigwedge_{p_i \in {}^\bullet t \setminus t^\bullet} v_i' = \text{FALSE} \\ & \wedge \bigwedge_{p_i \in t^\bullet} v_i' = \text{TRUE} \wedge \bigwedge_{p_i \in \mathcal{P} \setminus ({}^\bullet t \cup t^\bullet)} v_i' = v_i \end{aligned}$$

For t_1 in the Petri net in Figure 1, for example, we have:

$$\begin{aligned} a_{t_1} \triangleq & \\ & v_1 = \text{TRUE} \wedge v_1' = \text{FALSE} \wedge v_2' = \text{TRUE} \wedge v_3' = \text{TRUE} \\ & \wedge v_4' = v_4 \wedge v_5' = v_5 \wedge v_6' = v_6 \wedge v_7' = v_7 \end{aligned}$$

Accordingly, $\hat{a}_t$ is defined as follows:

$$\begin{aligned} \hat{a}_t & \triangleq a_t \vee \bigwedge_{p_i \in \mathcal{P}} v_i' = v_i \\ & \equiv \Big(\big(\bigwedge_{p_i \in {}^\bullet t} v_i \wedge \bigwedge_{p_i \in {}^\bullet t \setminus t^\bullet} \neg v_i' \wedge \bigwedge_{p_i \in t^\bullet} v_i'\big) \vee \bigwedge_{p_i \in {}^\bullet t \cup t^\bullet} v_i' = v_i\Big) \\ & \quad \wedge \bigwedge_{p_i \in \mathcal{P} \setminus ({}^\bullet t \cup t^\bullet)} v_i' = v_i \end{aligned}$$

To make the formula concise, $v =$ TRUE and $v =$ FALSE are denoted as v and $\neg v$. For the Petri net shown in Figure 1, for example, we have:

$$\begin{aligned} & \hat{a}_{t_1} \triangleq \\ & \Big((v_1 \wedge \neg v_1' \wedge v_2' \wedge v_3') \vee (v_1' = v_1 \wedge v_2' = v_2 \wedge v_3' = v_3)\Big) \\ & \wedge v_4' = v_4 \wedge v_5' = v_5 \wedge v_6' = v_6 \wedge v_7' = v_7 \end{aligned}$$

Below is the formula to be checked for the example Petri net and $k = 1$.

$$\begin{aligned} & I(V_0/V) \\ & \wedge\Big((v_{1,0} \wedge \neg v_{1,1} \wedge v_{2,1} \wedge v_{3,1}) \\ & \qquad \vee\big((v_{1,0} = v_{1,1}) \wedge (v_{2,0} = v_{2,1}) \wedge (v_{3,0} = v_{3,1})\big)\Big) \\ & \wedge\Big((v_{1,1} \wedge \neg v_{1,2} \wedge v_{4,2} \wedge v_{5,2}) \\ & \qquad \vee\big((v_{1,1} = v_{1,2}) \wedge (v_{4,0} = v_{4,2}) \wedge (v_{5,0} = v_{5,2})\big)\Big) \\ & \wedge\Big((v_{2,1} \wedge \neg v_{2,3} \wedge v_{6,3}) \vee \big((v_{2,1} = v_{2,3}) \wedge (v_{6,0} = v_{6,3})\big)\Big) \\ & \wedge\Big((v_{3,1} \wedge \neg v_{3,4} \wedge v_{7,4}) \vee \big((v_{3,1} = v_{3,4}) \wedge (v_{7,0} = v_{7,4})\big)\Big) \\ & \wedge\Big((v_{4,2} \wedge \neg v_{4,5} \wedge v_{6,5}) \vee \big((v_{4,2} = v_{4,5}) \wedge (v_{6,3} = v_{6,5})\big)\Big) \\ & \wedge\Big((v_{5,2} \wedge \neg v_{5,6} \wedge v_{7,6}) \vee \big((v_{5,2} = v_{5,6}) \wedge (v_{7,4} = v_{7,6})\big)\Big) \\ & \wedge\Big((v_{6,5} \wedge v_{7,6} \wedge \neg v_{6,7} \wedge \neg v_{7,7} \wedge v_{1,7}) \\ & \qquad \vee\big((v_{1,2} = v_{1,7}) \wedge (v_{6,5} = v_{6,7}) \wedge (v_{7,6} = v_{7,7})\big)\Big) \\ & \wedge P(v_{1,7}/v_1, v_{2,3}/v_2, v_{3,4}/v_3, v_{4,5}/v_4, v_{5,6}/v_5, v_{6,7}/v_6, \\ & \qquad v_{7,7}/v_7) \end{aligned}$$

This resulting formula is much smaller than Φ —the formula used in ordinary bounded model checking. a_t contains at least $m(= |\mathcal{P}|)$ literals, while its counterpart in our approach (that is, $\hat{a}_t$ with some terms removed) only contains at most $4|{}^\bullet t| + 3|t^\bullet|$ literals. The difference becomes larger if the number of places that are neither input nor output places increases for each transition.

It should also be noted that in this particular case, $k = 1$ is enough to explore all the reachable states. On the other hand, ordinary bounded model checking needs $k = 3$.

Since only boolean values are involved, a propositional SAT solver can be used to determine the satisfiability. In [9], Yu, Ciardo, and Lüttgen conducted a comprehensive comparison with their BDD-based model checking approach with two SAT-based bounded model checking methods: ours [7] and that by Helajanko and Niemelä [2]. The results show that different methods work best for different problems and that there is no clear winner.

5. Systems with Integers

A problem with proportional SAT based model checking is that software programs generally deal with integer variables rather than boolean variables. Because in ordinary symbolic model checking, all data types are encoded as boolean variables, it is hard to handle integer variables, especially unbounded ones.

In this section, we propose a bounded model checking method that can handle integer variable directly to achieve efficient verification for such software systems.

To represent such systems, we use unbounded integer variables. To make the formula decidable, we limit any formula specifying the system to boolean combinations of linear arithmetic formulas. We define a linear arithmetic formula as a formula of the form:

$$\sum_{x_i} c_i^0 x_i + d^0 \ OP \ \sum_{x_i} c_j^1 x_i + d^1 \quad (\forall j, x_j \in \mathcal{Z})$$

where x_i is an integer variable and c_i^0, c_i^1 are integer coefficients, and OP is one of the binary operators: $<, \leqq, =, >, \geqq$.

Figure 2 represents a program for the bounded-buffer producer-consumer problem. This problem is a well know concurrency problem and illustrates the need for synchronization in systems where many processes share a resource. This particular program is an instance with one producer and one consumer. The action $a_{produce}$ produces an item and puts it in the buffer, while the other action $a_{consume}$ consumes an item from the buffer.

This system has a parameterized constant *size* which determines the size of the buffer. This system should be correctly work for any value of *size*. For simplicity, we assume that *size* is just a positive integer in this paper; it is easy to perform parameterized model checking, though. As a correctness criteria, consider that $produced - consumed = size - available$ must hold invariantly. This property can be verified by checking reachability of states where the following predicate holds:

$$P \triangleq \neg(produced - consumed = size - available)$$

Figures 3 and 4 show respectively $\hat{\Phi}_2$ and $\hat{\Phi}_2$ with some terms removed the way described in Section 3.

Off-the-shelf SMT solvers, such as Yices [1], can be used to check satisfiability. Performance results will soon be reported.

6. Conclusion

In this paper, we proposed a new bounded model checking method for verification of concurrent systems. By exploiting concurrency, our method generates much more succinct formulas than ordinary bounded model checking. We applied the proposed approach to safe Petri nets and concurrent systems with unbounded integers.

There are many possible directions for extending the proposed methods.

a) Transition ordering

In this paper we did not discuss how O, the function for ordering actions, should be determined. In fact the performance of the proposed model checking approach depends critically on this ordering function. As for safe Petri nets, we proposed a heuretic algorithm for this task in [7]. The algorithm traverses the net structure in a depth-first manner and puts early encountered transitions forward.

b) Model checking against temporal logic formulas

Our approach can be extended so that a general class of temporal logic formulas can be verified. For example, in [6] an extension that allows verification of a subset of LTL$_{-X}$ formulas is proposed.

c) Unbounded model checking

It is known that bounded model checking can be extended to unbounded model checking by making use of Craig's interpolants [5]. When only Boolean variables are involved, such as the case for safe Petri nets, interpolant-based unbounded model checking can be adapted to our semantics. In [4] we show that our approach significantly improves the performance of verification through the application to feature interaction detection for telecommunication services.

Acknowledgments

We would like to thank Takahiro Tanaka for his work in our laboratory. This research was supported in part by the MEXT Global COE program (Center of Excellence for Founding Ambient Information Society Infrastructure).

References

[1] B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proc. of 18th Conf. on Computer Aided Verification (CAV 2006)*, volume 4144 of *LNCS*, pages 81–94, Seattle, USA, Aug. 2006. Springer.
[2] K. Heljanko and I. Niemelä. Answer set programming and bounded model checking. In A. Provetti and T. C. Son, editors, *Answer Set Programming*, 2001.
[3] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
[4] T. Matsuo, T. Tsuchiya, and T. Kikuno. Feature interaction verification using unbounded model checking with interpolation. *IEICE Transactions on Information and Systems*, E92-D(6), 6 2009.
[5] K. L. McMillan. Applications of craig interpolants in model checking. In N. Halbwachs and L. D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2005.
[6] S. Ogata. PhD thesis, Dept. of Info. Syst. Eng., Osaka Univ.
[7] S. Ogata, T. Tsuchiya, and T. Kikuno. Sat-based verification of safe petri nets. In *Proceedings of 2nd International Symposium on Automated Technology for Verification and Analysis (ATVA 2004)*, pages 72–92, 11 2004.
[8] D. Peled and T. Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters*, 63(5):243–246, 1997.
[9] A. J. Yu, G. Ciardo, and G. Lüttgen. Decision-diagram-based techniques for bounded reachability checking of asynchronous systems. *Journal on Software Tools for Technology Transfer*, 11(2):117–131, 2009.

$$
\begin{array}{lll}
V & \triangleq & \{\mathit{available}, \mathit{produced}, \mathit{consumed}\} \\
I & \triangleq & \mathit{produced} = 0 \wedge \mathit{consumed} = 0 \wedge \mathit{available} = \mathit{size} \\
A & \triangleq & \{ \\
& a_{produce} :: & 0 < \mathit{available} \wedge \mathit{produced}' = \mathit{produced} + 1 \wedge \mathit{available}' = \mathit{available} - 1 \\
& a_{consume} :: & \mathit{available} < \mathit{size} \wedge \mathit{consumed}' = \mathit{consumed} + 1 \wedge \mathit{available}' = \mathit{available} + 1 \\
\}
\end{array}
$$

Figure 2 A program for bounded-buffer

$$
\begin{array}{lr}
\mathit{produced}_0 = 0 \wedge \mathit{consumed}_0 = 0 \wedge \mathit{available}_0 = \mathit{size} & I(V_0/V) \\
\wedge\, ((0 < \mathit{available}_0 \wedge \mathit{produced}_1 = \mathit{produced}_0 + 1 \wedge \mathit{available}_1 = \mathit{available}_0 - 1) & \\
\qquad \vee\, (\mathit{produced}_1 = \mathit{produced}_0 \wedge \mathit{available}_1 = \mathit{available}_0)) & \\
\quad \wedge\, (\mathit{consumed}_1 = \mathit{consumed}_0) & \hat{a}_{produce}(V_0/V, V_1/V') \\
\wedge\, ((\mathit{available}_1 < \mathit{size} \wedge \mathit{consumed}_2 = \mathit{consumed}_1 + 1 \wedge \mathit{available}_2 = \mathit{available}_1 + 1) & \\
\qquad \vee\, (\mathit{consumed}_2 = \mathit{consumed}_1 \wedge \mathit{available}_2 = \mathit{available}_1)) & \\
\quad \wedge\, (\mathit{produced}_2 = \mathit{produced}_1) & \hat{a}_{consume}(V_1/V, V_2/V') \\
\wedge\, ((0 < \mathit{available}_2 \wedge \mathit{produced}_3 = \mathit{produced}_2 + 1 \wedge \mathit{available}_3 = \mathit{available}_2 - 1) & \\
\qquad \vee\, (\mathit{produced}_3 = \mathit{produced}_2 \wedge \mathit{available}_3 = \mathit{available}_2)) & \\
\quad \wedge\, (\mathit{consumed}_3 = \mathit{consumed}_2) & \hat{a}_{produce}(V_2/V, V_3/V') \\
\wedge\, ((\mathit{available}_3 < \mathit{size} \wedge \mathit{consumed}_4 = \mathit{consumed}_3 + 1 \wedge \mathit{available}_4 = \mathit{available}_3 + 1) & \\
\qquad \vee\, (\mathit{consumed}_4 = \mathit{consumed}_3 \wedge \mathit{available}_4 = \mathit{available}_3)) & \\
\quad \wedge\, (\mathit{produced}_4 = \mathit{produced}_3) & \hat{a}_{consume}(V_3/V, V_4/V') \\
\wedge\, \neg(\mathit{produced}_4 - \mathit{consumed}_4 = \mathit{size} - \mathit{available}_4) & P(V_4/V)
\end{array}
$$

Figure 3 $\hat{\Phi}_2$ for the bounded-buffer program ($k = 2, n = 2$)

$$
\begin{array}{lr}
\mathit{produced}_0 = 0 \wedge \mathit{consumed}_0 = 0 \wedge \mathit{available}_0 = \mathit{size} & I(V_0/V) \\
\wedge\, ((0 < \mathit{available}_0 \wedge \mathit{produced}_1 = \mathit{produced}_0 + 1 \wedge \mathit{available}_1 = \mathit{available}_0 - 1) & \\
\qquad \vee\, (\mathit{produced}_1 = \mathit{produced}_0 \wedge \mathit{available}_1 = \mathit{available}_0)) & \hat{a}_{produce}(V_0/V, V_1/V') \\
\wedge\, ((\mathit{available}_1 < \mathit{size} \wedge \mathit{consumed}_2 = \mathit{consumed}_0 + 1 \wedge \mathit{available}_2 = \mathit{available}_1 + 1) & \\
\qquad \vee\, (\mathit{consumed}_2 = \mathit{consumed}_0 \wedge \mathit{available}_2 = \mathit{available}_1)) & \hat{a}_{consume}(V_1/V, V_2/V') \\
\wedge\, ((0 < \mathit{available}_2 \wedge \mathit{produced}_3 = \mathit{produced}_1 + 1 \wedge \mathit{available}_3 = \mathit{available}_2 - 1) & \\
\qquad \vee\, (\mathit{produced}_3 = \mathit{produced}_1 \wedge \mathit{available}_3 = \mathit{available}_2)) & \hat{a}_{produce}(V_2/V, V_3/V') \\
\wedge\, ((\mathit{available}_3 < \mathit{size} \wedge \mathit{consumed}_4 = \mathit{consumed}_2 + 1 \wedge \mathit{available}_4 = \mathit{available}_3 + 1) & \\
\qquad \vee\, (\mathit{consumed}_4 = \mathit{consumed}_2 \wedge \mathit{available}_4 = \mathit{available}_3)) & \hat{a}_{consume}(V_3/V, V_4/V') \\
\wedge\, \neg(\mathit{produced}_3 - \mathit{consumed}_4 = \mathit{size} - \mathit{available}_4) & P(V_4/V)
\end{array}
$$

Figure 4 A concise formula derived from $\hat{\Phi}_2$