BDD vs. Constraint-Based Model Checking: An Experimental Evaluation for Asynchronous Concurrent Systems

Tevfik Bultan

Department of Computer Science
University of California
Santa Barbara, CA 93106, USA
bultan@cs.ucsb.edu

Abstract. BDD-based symbolic model checking has been applied to a wide range of applications. Recently, constraint-based approaches, which use arithmetic constraints as a symbolic representation, have been used in symbolic model checking of infinite-state systems. We argue that use of constraint-based model checking is not limited to infinite-state systems. It can also be an efficient alternative to BDD-based model checking for systems with integer variables that have large domains. In this paper we compare the performance of BDD-based model checker SMV to the performance of our constraint-based model checker on verification of several asynchronous concurrent systems. The results indicate that constraint-based model checking is a viable option for verification of asynchronous concurrent systems that require more than 5 boolean variables to encode each integer variable.

1 Introduction

Model checking has been used in verification of diverse applications ranging from hardware protocols [McM93] to software specifications [CAB+98]. The success of model checking has been partially due to use of efficient data structures like Binary Decision Diagrams (BDDs) which can encode boolean functions in a highly compact format [Bry86]. The main idea in BDD-based *symbolic* model checking is to represent sets of system states and transitions as boolean logic formulas, and manipulate them efficiently using the BDD data structure [BCM+90].

An important property of the BDD data structure is that it supports operations such as intersection, union, complement, equivalence checking and existential quantifier elimination (used to implement relational image computations)—which also happen to be the main operations required for model checking. However, an efficient encoding for boolean domains may not be efficient for all variable types. For example, BDD-based model checkers can be very inefficient in representing arithmetic constraints [CAB+98].

This work was supported in part by NSF grant CCR-9970976 and a University of California Regents' Junior Faculty Fellowship.

Another shortcoming of the BDD representation is its inability to encode infinite domains. Without abstraction, BDDs cannot be used for analyzing infinite-state systems—even those with just one unbounded integer. BDDs encode all underlying datatypes as boolean variables; hence all BDD-based model checkers inherently require the underlying types to be bounded.

Recently, arithmetic constraints have been used as a symbolic representation in model checking [AHH96,BGP97]. For example, HyTech, a symbolic model checker for hybrid systems, encodes real domains using linear constraints on real variables [AHH96]. We developed a model checker for integer based systems which uses Presburger arithmetic (integer arithmetic without multiplication) constraints as its underlying state representation [BGP97,BGP]. Our model checker uses the Omega library [KMP+95] to manipulate Presburger arithmetic constraints. In [DP99] model checking queries are converted into constraint logic programs, and a CLP(R) library is used to verify concurrent systems by mapping integer variables to real domains.

Constraint representations make it possible to verify infinite-state systems since they can represent variables with infinite domains. There are algorithms for intersection, union, complement, equivalence checking and existential quantifier elimination for both real and integer constraint representations mentioned above. However model checking becomes undecidable for infinite-state systems. Hence the fixpoint computations are not guaranteed to converge. This problem is addressed using conservative approximation techniques [BGP] which guarantee convergence but do not guarantee a definite answer, i.e., the model checker 1) may report that the property is verified, 2) provide a counter-example demonstrating violation of the property, or 3) report that the analysis is inconclusive.

Using arithmetic constraints one can also represent variables with finite domains. We just have to add additional constraints which show the range of values that an integer variable can take. An interesting issue is, then, comparing the performance of BDD-based model checking to constraint-based model checking for finite-state systems with integer variables.

In this paper we compare the performance of a BDD-based model checker (SMV [McM93]) and a constraint-based model checker (our model checker based on Omega library [BGP97,BGP]) in verification of asynchronous concurrent systems with integer variables. As we mentioned above, constraint-based model checkers are able to handle integer variables with finite domains. However on the extreme case where integer variables can take only two values, they can be treated as boolean variables and represented using BDDs. Using a constraint-representation would be very inefficient in such a case. On the other hand, although BDD-based model checkers are not capable of handling systems with unbounded integers, if the variables are restricted to a finite set of values, they can be represented using a set of boolean variables using a binary encoding. Our goal in this paper is to investigate the middle ground between these two extremes where the integer variables are neither unbounded nor have only two possible valuations.

We perceive efforts in constraint-based model checking as not only a way to solve infinite-state verification problems, but also as a way to deal with problems with large variable domains using formalisms that are more expressive than boolean logic formulas. However, because of the added expressive power, manipulation algorithms for these formalisms have higher complexity than corresponding algorithms for BDDs. These powerful algorithms may not be worthwhile to use for small domains because of their high complexity. On the other hand, for large domains their complexity maybe justified. In this paper we investigate the following question experimentally: When is the use of integer constraint representation justified instead of BDD encodings? We investigate this issue on verification of asynchronous concurrent systems.

2 Related Work

Another approach to infinite-state model checking is to use automata-based representations. Automata can be used to represent arithmetic constraints on unbounded integer variables [WB95,BKR96,KSA98]. An arithmetic constraint on k integer variables is represented by a k-track automata that accepts a string if it corresponds to a k-dimensional integer vector (in binary representation) that satisfies the corresponding arithmetic constraint. Again, since the automata representation supports the necessary operations, it can be used in symbolic model checking.

The constraint and automata-based representations provide two different ways of implementing model checking computations for systems with unbounded integer variables. In [SKR98] these two approaches are compared experimentally for reachability analysis of several concurrent systems. The results show no clear winner. On some problem instances the constraint representation is superior, on some others automata representation is better. In automata-based representations, restricting variables to fixed finite domains ends up converting the automata representation to a model isomorphic to BDDs [KSA98]. Hence for the experiments we conduct in this paper the automata-based representation is not an alternative (or to put it another way, it is equivalent to BDD-based model checking).

Using a BDD-based model checker such as SMV [McM93] for checking systems with integer variables can easily result in inefficient encodings of arithmetic constraints [CAB+98]. In [CABN97], Chan *et al.* report that representing integers using bitwise BDD representations is not efficient when the input system contains non-linear constraints. They present a technique in which SMV is augmented with a constraint solver. However, because of the restrictions they put on the types of systems that can be analyzed, their technique is not applicable to the systems we analyze in this paper.

Another approach to dealing with integer variables in BDD-based model checking is to use abstractions. In [CGL92], a conservative abstraction method is presented for model-checking infinite-state programs. The main idea is to produce a finite model of the program using a suitable abstraction technique (e.g., congruence modulo an integer, single-bit abstraction, symbolic abstraction), and then to check the property of interest on the abstraction. For systems such as the ones we analyze in this paper, finding a good abstraction maybe as difficult as proving the invariants of the system. On the other hand, automated abstractions such as the ones presented in [HKL+98] are not strong enough to eliminate the integer variables in the systems we analyze in this paper.

3 Symbolic Model Checking

In model checking, the system to be analyzed is represented as a transition system $TS = (S, I, R)$ with a set of states S, a set of initial states $I \subseteq S$, and a transition relation $R \subseteq S \times S$. The transition system model is never explicitly generated in symbolic model checking. For example, BDD-based model checkers represent transition relation R as a set of boolean logic formulas.

A popular temporal logic for specifying temporal properties of transition systems is Computation Tree Logic (CTL) [CES86] which consists of a set of temporal operators (the next-state operators EX and AX, the until operators EU and AU, the invariant operators EG and AG, and the eventuality operators EF and AF) for specifying temporal properties.

Our goal in model checking a system $TS = (S, I, R)$ and a temporal property p is (we use p to denote its truth set) : 1) either to prove that the system TS satisfies the property p by showing that $I \subseteq p$, or 2) to demonstrate a bug by finding a state $s \in I \cap \neg p$, and generating a counter-example path starting from s.

Assume that there exists a representation for sets of states which supports tests for equivalence and membership. Then, if we can represent the truth set of the temporal property p, and the set of initial states I using this representation, we can check the two conditions listed above. If the state space is finite, explicit state enumeration would be one such representation. Note that as the state spaces of the programs grow, explicit state enumeration will become more expensive since the size of this representation is linearly related to the number of states in the set it represents. Unfortunately, state space of programs increase exponentially with the number of variables and concurrent components. This state space explosion problem makes a naive implementation of the explicit state enumeration infeasible.

Another approach is to use a *symbolic representation* for encoding sets of states. For example, a logic formula which is semantically interpreted as a set of states, can be used as a symbolic representation. Boolean logic formulas (stored using the BDD data structure) are the most common symbolic representation used in model checking [BCM+90]. Recently, we used Presburger arithmetic (integer arithmetic without multiplication) formulas for the same purpose [BGP97,BGP].

Model checking procedures use state space exploration to compute the set of states which satisfy a temporal property. Fixpoints corresponding to truth sets of temporal formulas can be computed by iteratively aggregating states using pre-condition computations (which correspond to the next state operator EX). Temporal properties which require more than one fixpoint computation can be computed recursively starting from the inner fixpoints and propagating the partial results to the outer fixpoints.

All temporal properties in CTL can be expressed using boolean connectives, next state operator EX, and least fixpoints. For example, EF$p \equiv \mu x \,.\, p \vee \text{EX}\, x$. The least fixpoint of a monotonic functional can be computed by starting from the bottom element (i.e., **false** $\equiv \emptyset$) and by iteratively applying the functional until a fixpoint is reached.

Assume that `Symbolic` is the data type used for encoding sets of states. In order to implement a symbolic model checker based on `Symbolic` data type we need the following procedures:

`Symbolic Not(Symbolic)` : Given an argument that represents a set $p \subseteq S$, it returns a representation for $S - p$.
`Symbolic And(Symbolic,Symbolic)` : Given two arguments representing two sets $p, q \subseteq S$, it returns a representation for $p \cap q$.
`Symbolic Or(Symbolic,Symbolic)` : Given two arguments representing two sets $p, q \subseteq S$, it returns a representation for $p \cup q$.
`Symbolic EX(Symbolic)` : Given an argument that represents a set $p \subseteq S$, it returns a representation for the set $\{s \mid \exists s' \,.\, s' \in p \ \wedge \ (s, s') \in R\}$.
`Boolean Equivalent(Symbolic, Symbolic)` : Given two arguments representing two sets $p, q \subseteq S$, it returns **true** if $p \equiv q$, returns **false** otherwise.

Using the procedures described above, given a temporal formula, we can compute its truth set by computing the fixpoint that corresponds to that temporal formula.

The computation of the procedure `EX` involves computing a relational image. Given a set $p \subseteq S$ and a relation $X \subseteq S \times S$ we use $X\ p$ to denote relational image of p under X, i.e., $X\ p$ is defined as restricting the domain of X to set p, and returning the range of the result. Note that we can think of relation X as a functional $X\ :\ 2^S \rightarrow 2^S$. Then, $X\ p$ denotes the application of the functional X to set p.

Let R^{-1} denote the inverse of the transition relation R. Then EX $p \equiv R^{-1}\ p$, i.e., functional EX corresponds to the inverse of the transition relation R. Hence, we can compute the procedure `EX` using a relational image computation. Most model checkers represent transition relation R in a partitioned form to make the relational image computation more efficient [BCL91].

Any representation which is able to encode the set of initial states I and the set of atomic properties AP, and supports the above functionality can be used as a symbolic representation in a model checker. We call such a representation an *adequate language* for model checking [KMM+97]. For example, for finite-state systems, boolean logic would be one such representation. It is possible to implement procedures for negation, conjunction, disjunction and equivalence checking of boolean logic formulas. If we can represent the transition relation R as a boolean logic formula, then relational image computation $R^{-1}\ p$ can be computed by conjuncting the formula representing R^{-1} and the formula representing p, and then eliminating the variables in the domain of the resulting relation using existential quantifier elimination. BDDs are an efficient data structure for representing boolean logic formulas, and they support all the functionality mentioned above [Bry86]. They have been successfully used for model checking [BCM$^+$90,McM93]. However, they can not encode infinite variable domains.

Recently, we developed a model checker for systems with unbounded integer variables using Presburger arithmetic formulas as a symbolic representation [BGP97]. There are effective procedures for manipulating Presburger formulas which support the above functionality—for example Omega Library implements a set of such procedures [KMP$^+$95]. We implemented a model checker using Omega Library as our symbolic manipulator. However, model checking computations become undecidable for infinite domains, i.e., the fixpoint computations corresponding to temporal properties may not always converge for infinite domains. We addressed this issue in [BGP] using conservative approximations.

4 Example Concurrent Systems

The examples we use in this paper have the following characteristics: 1) they are all asynchronous concurrent systems, and 2) they all use shared integer variables to control their synchronization. We think these type of systems are especially suitable for constraint-based representations. Most of our examples are from [And91].

We represent each concurrent system with a set of events, where each event is considered atomic. The state of a program is determined by the values of its data and control variables. If a variable v is used in an event, then the symbol v' denotes the new value of v after the action is taken. If v' is not mentioned in an event, then we assume that its value is not altered by that event. Each event specification defines a transition relation over the Cartesian product of the domains of the variables in the system. The transition relation of the overall concurrent system is defined as the union of the transition relations of all events in the system.

Program: Bakery
Data Variables: a, b: positive integer
Control Variables: $pc_1 : \{T_1, W_1, C_1\}$, $pc_2 : \{T_2, W_2, C_2\}$
Initial Condition: $a = b = 0 \ \wedge \ pc_1 = T_1 \ \wedge \ pc_2 = T_2$
Events:
$e_{T_1} \ : pc_1 = T_1 \ \wedge \ pc_1' = W_1 \ \wedge \ a' = b + 1$
$e_{W_1} : pc_1 = W_1 \ \wedge \ (a < b \ \vee \ b = 0) \ \wedge \ pc_1' = C_1$
$e_{C_1} \ : pc_1 = C_1 \ \wedge \ pc_1' = T_1 \ \wedge \ a' = 0$
$e_{T_2} \ : pc_2 = T_2 \ \wedge \ pc_2' = W_2 \ \wedge \ b' = a + 1$
$e_{W_2} : pc_2 = W_2 \ \wedge \ (b < a \ \vee \ a = 0) \ \wedge \ pc_2' = C_2$
$e_{C_2} \ : pc_2 = C_2 \ \wedge \ pc_2' = T_2 \ \wedge \ b' = 0$

Bakery algorithm shown above for two processes is a mutual exclusion algorithm. The algorithm we present above is the coarse grain solution [And91] which can be further refined to implement without fetch-and-add instructions.

Below we show a solution to sleeping barber problem [And91]. The barber allows a new customer into the shop with the event e_{next_1}. The customer gets a chair by calling the event $e_{haircut_1}$ (as long as their is an available chair). Then the barber starts the haircut with event e_{next_2}. When the haircut is finished the barber executes e_{done_1}, and waits (e_{done_2}) till the customer leaves by executing the event $e_{haircut_2}$.

Program: Barber
Data Variables: $cinchair, cleave, bavail, bbusy, bdone$: positive integer
Control Variables: $pc_1 : \{1, 2\}$, $pc_2 : \{1, 2\}$ $pc_3 : \{1, 2\}$
Initial Condition: $cinchair = cleave = bavail = bbusy = bdone = 0$
$\wedge\ pc_1 = 1\ \wedge\ pc_2 = 1\ \wedge\ pc_3 = 1$
Events:
$e_{haircut_1} : pc_1 = 1\ \wedge\ pc_1' = 2\ \wedge\ cinchair < bavail\ \wedge\ cinchair' = cinchair + 1$
$e_{haircut_2} : pc_1 = 2\ \wedge\ pc_1' = 1\ \wedge\ cleave < bdone\ \wedge\ cleave' = cleave + 1$
$e_{next_1} : pc_2 = 1\ \wedge\ pc_2' = 2\ \wedge\ bavail' = bavail + 1$
$e_{next_2} : pc_2 = 2\ \wedge\ pc_2' = 1\ \wedge\ bbusy < cinchair\ \wedge\ bbusy' = bbusy + 1$
$e_{finish_1} : pc_3 = 1\ \wedge\ pc_3' = 2\ \wedge\ bdone < bbusy\ \wedge\ bdone' = bdone + 1$
$e_{finish_2} : pc_3 = 2\ \wedge\ pc_3' = 1\ \wedge\ bdone = cleave$

Program: Readers-Writers
Data Variables: nr, nw: positive integer
Initial Condition: $nr = nw = 0$
Events:
$e_{reader-enter} : nw = 0\ \wedge\ nr' = nr + 1$
$e_{reader-exit} : nr > 0\ \wedge\ nr' = nr - 1$
$e_{writer-enter} : nr = 0\ \wedge\ nw = 0\ \wedge\ nw' = nw + 1$
$e_{writer-exit} : nw > 0\ \wedge\ nw' = nw - 1$

Above, a well-known algorithm for readers-writers problem is presented. The invariant of the readers-writers problem states that at any time there would be either no writers accessing the database or no readers, and the number of writers should never be more than one.

Program: Bounded-Buffer
Parameterized Constant: $size$: positive integer
Data Variables: $available, produced, consumed$: positive integer
Initial Condition: $produced = consumed = 0\ \wedge\ available = size$
Events:
$e_{produce} : 0 < available\ \wedge\ produced' = produced + 1$
$\wedge\ available' = available - 1$
$e_{consume} : available < size\ \wedge\ consumed' = consumed + 1$
$\wedge\ available' = available + 1$

Program: Circular-Queue
Parameterized Constant: $size$: positive integer
Data Variables: $occupied, head, tail, produced, consumed$: positive integer
Initial Condition: $occupied = head = tail = produced = consumed = 0$
Events:

$$e_{produce} : occupied < size \ \wedge \ occupied' = occupied + 1 \ \wedge \ produced' = produced + 1$$
$$\wedge \ (tail = size \ \wedge \ tail' = 0 \ \vee \ tail < size \ \wedge \ tail' = tail + 1)$$
$$e_{consume} : occupied > 0 \ \wedge \ occupied' = occupied - 1 \ \wedge \ consumed' = consumed + 1$$
$$\wedge \ (head = size \ \wedge \ head' = 0 \ \vee \ head < size \ \wedge \ head' = head + 1)$$

Two algorithms given above present bounded-buffer implementations. Both these systems have a parameterized constant $size$ which specifies the size of the buffer. Since $size$ is parameterized the systems given above should be correct for any value of $size$.

In Table 1 we list the invariants the systems presented above have to satisfy.

5 Experimental Evaluation

We translated the examples given above to the SMV input language. For each concurrent process we used the `process` declaration in SMV input language which supports asynchronous composition. SMV converts all integer variables to a binary representation since it is a BDD-based model checker. To represent the parameterized constant $size$ in the bounded-buffer and circular-queue systems we used an uninitialized variable which always preserves its value.

Our omega library based model checker (OMC) accepts a Presburger arithmetic formula for each event in the input system. It forms the global transition relation by combining these formulas disjunctively. It uses asynchronous composition to combine two concurrent components. It is not efficient to map variables with small domains (such as program counters) to integer variables. So, for the examples with control variables we used control point partitioning to eliminate the control variables [BGP].

To compare the performances of SMV and OMC (Omega library Model Checker) we assigned a finite domain to each integer variable. We generated several instances for each concurrent system by restricting the integer variables to different ranges. We started with a range of $0 \le i < 8$ for each integer variable i (which makes it possible to represent each variable i with 3 boolean variables in SMV). We generated a different problem instance for each additional boolean variable until 9 boolean variables are required to encode each integer variable (i.e., the largest range we used is $0 \le i < 512$).

In Figures 1 and 2 we show the performances of both SMV and OMC in terms of executions time and memory usage. We ran all our experiments on an Intel Pentium III PC (500MHz, 128 MByte main memory) running Solaris. Each graph shows experiments on one concurrent system. Data points in each individual graph is generated by only changing the range of values that integer variables are allowed to take. The x axis in these graphs show the number of boolean variables required for the binary encoding of each integer variable (which ranged from 3 to 9 in our experiments). So, for each point in the graph, the range of each integer variable i in the concurrent system verified in that particular experiment is $0 \leq i < 2^x$.

Figure 1 shows the performances of SMV and OMC on verification of both two and three process implementations of the bakery algorithm with respect to the property BAKERY shown in Table 1. Both execution time and memory usage for SMV shows an exponential increase with increasing number of boolean variables used in encoding the integer variables. However, the performance of OMC stays constant with respect to increasing variable domains. This is because of the fact that the complexity of the procedures used in OMC are in terms of the number of constraints used in the constraint representation. Changing the maximum value that an integer variable can take from one integer constant to another integer constant does not increase the number of constraints in the constraint representation. Therefore the performance of OMC does not deteriorate with increasing domains of integer variables. On the other hand, the worst-case complexity of the BDD representation is exponential in the number of boolean variables it represents. This explains the exponential increase in execution time and memory usage of SMV with the increasing number of boolean variables used in the encoding. SMV outperforms OMC if the integer variables can be represented with fewer than 5 boolean variables for this problem instance.

The performance of both tools deteriorate significantly if the number of processes is increased. However, the cost of constraint-based model checking seems to increase more significantly compared to BDD-based model checking.

Figure 1 shows the performances of SMV and OMC on verification of the sleeping barber problem solution given above with respect to the property BARBER shown in Table 1. Again the execution time and memory usage of SMV increases exponentially with the increasing number of boolean variables used to encode the integer variables. An interesting result is that the cutoff point between the performances of two model-checkers is around 5 again.

Figure 1 shows the results for the readers-writers system given above with respect to the property READERS-WRITERS given in Table 1. The performances of SMV and OMC are similar to their relative performances for BAKERY and BARBER in terms of execution time. Again the cutoff point is around 5. For memory usage SMV seems to perform consistently worse. This is due to the fact that SMV allocates a constant amount of memory (which is around 1MByte) even for the smallest problem instances. Hence the constant distance between two memory usage curves is not important. The important fact is, as seen at the right end of the graph, memory usage of SMV starts increasing when the ranges of the integer variables keep increasing.

We verified the bounded-buffer algorithm with respect to the property BOUNDED-BUFFER shown in Table 1. We restricted to domain of parameterized constant $size$ to two possible ranges $0 \leq size < 8$ and $0 \leq size < 16$. Figure 2 shows the performance of SMV and OMC with respect to increasing domains of integer variables. Again the performance of OMC stays constant for all instances. The execution time of SMV increases exponentially whereas its memory requirement increases linearly after a certain threshold. Again the cut-off point between the performances of SMV and OMC is around the point where 5 boolean variables are required to encode each integer variable.

Finally, we verified the circular-queue algorithm with respect to the property CIRCULAR-QUEUE shown in Table 1. Similar to bounded-buffer, we used two possible ranges for parameterized constant *size*. Relative performances of SMV and OMC for circular-queue are similar to their relative performances for bounded-buffer. The only difference is the fact that circular-queue requires more execution time and memory usage from both SMV and OMC.

Based on our experiments we conclude that the constraint-based model checking techniques can be justified for systems with integer variables which require more than 5 boolean variables to encode. For any smaller domain it may not be worthwhile to use constraint-based approach, a BDD-based model checker could give a better performance.

In Table 2 we show the performance of SMV and OMC around this cutoff point, i.e., for the problem instances in Table 2 each integer variable i is restricted to the range $0 \leq i < 32$ (we also restricted the parameterized constant $size$ to $0 \leq size < 16$). As shown in Table 2 SMV and OMC have similar performances. However for the BAKERY the increase in execution time and memory usage of OMC with respect to increasing number of processes is significantly higher compared to SMV. For 4 processes OMC did not converge in one hour (we indicate this with $\uparrow$ in Table 2).

Another shortcoming of OMC is demonstrated in the verification of properties BARBER-1, BOUNDED-BUFFER-3 and CIRCULAR-QUEUE-1 shown in Table 1. None of the fixpoint computations for these properties converged (in an hour) when we tried to verify them using OMC. This is the price we pay for using an expressive representation such as constraints. Although the domain is finite, if the fixpoint computations take too many iterations the number of constraints in the representation increases significantly which makes the fixpoint iterations very expensive.

Note that BARBER-1 is a subclause of the invariant BARBER, i.e., BARBER $\rightarrow$ BARBER-1. Similarly, BOUNDED-BUFFER $\rightarrow$ BOUNDED-BUFFER-3 and CIRCULAR-QUEUE $\rightarrow$ CIRCULAR-QUEUE-1. Although OMC is able to verify stronger invariants BARBER, BOUNDED-BUFFER and CIRCULAR-QUEUE, it is not able to verify BARBER-1, BOUNDED-BUFFER-3 and CIRCULAR-QUEUE-1. This shows a characteristic of constraint-based model checking which is similar to theorem-proving, if we give more information about the system by providing a stronger invariant, constraint-based model checker is more likely to succeed.

Based on the results in Table 2 it is reasonable to say that SMV is a better alternative to OMC if the integer variables can be represented with 5 or less boolean variables per integer variable. On the other hand based on the results shown in Figures 1 and 2 we can say that constraint-based model checking is a valuable tool for verifying asynchronous concurrent systems with integer variables which have finite but large domains. Also note that the constraint-based model checkers have the additional benefit of being able to verify infinite-state systems. In fact, for all the instances other than properties BARBER-1, BOUNDED-BUFFER-3 and CIRCULAR-QUEUE-1, OMC is able to verify the infinite-state versions of all the systems presented in this paper.

Table 1. List of problem instances used in the experiments

Problem Instance	Property
BAKERY	$\mathrm{AG}(\neg(pc_1 = C_1 \ \wedge\ pc_2 = C_2))$
BARBER	$\mathrm{AG}(cinchair \geq cleave \ \wedge\ bavail \geq bbusy \geq bdone$ $\wedge\ cinchair \leq bavail \ \wedge\ bbusy \leq cinchair \ \wedge\ cleave \leq bdone)$
BARBER-1	$\mathrm{AG}(cinchair \geq cleave \ \wedge\ bavail \geq bbusy \geq bdone)$
BARBER-2	$\mathrm{AG}(cinchair \leq bavail \ \wedge\ bbusy \leq cinchair)$
BARBER-3	$\mathrm{AG}(cleave \leq bdone)$
READERS-WRITERS	$\mathrm{AG}((nr = 0 \ \vee\ nw = 0) \ \wedge\ nw \leq 1)$
BOUNDED-BUFFER	$\mathrm{AG}(produced - consumed = size - available$ $\wedge\ 0 \leq available \leq size)$
BOUNDED-BUFFER-1	$\mathrm{AG}(produced - consumed = size - available)$
BOUNDED-BUFFER-2	$\mathrm{AG}(0 \leq available \leq size)$
BOUNDED-BUFFER-3	$\mathrm{AG}(0 \leq produced - consumed \leq size)$
CIRCULAR-QUEUE	$\mathrm{AG}(0 \leq produced - consumed \leq size$ $\wedge\ produced - consumed = occupied)$
CIRCULAR-QUEUE-1	$\mathrm{AG}(0 \leq produced - consumed \leq size)$
CIRCULAR-QUEUE-2	$\mathrm{AG}(produced - consumed = occupied)$

6 Conclusions

Experimental results we obtain in this work suggests that constraint-based model checking can be more efficient than BDD-based model checking for verification of asynchronous concurrent systems if more than 5 boolean variables are required to encode each integer variable. This supports our view that constraint-based model checking is not limited to infinite-state systems but can also be useful for verification of systems with large integer domains.

In the future we would like to compare the performance of constraint-based model checking with the performance of word-level model checking [CZ95]. We are also planning to investigate the performance of our composite model checking approach [BGL98] with respect to BDD-based representations.

We would also like to investigate the complexity analysis of both BDD and constraint-based model checking for the type of systems analyzed in this paper.

References

[AHH96] R. Alur, T. A. Henzinger, and P. Ho. Automatic symbolic verification of em-

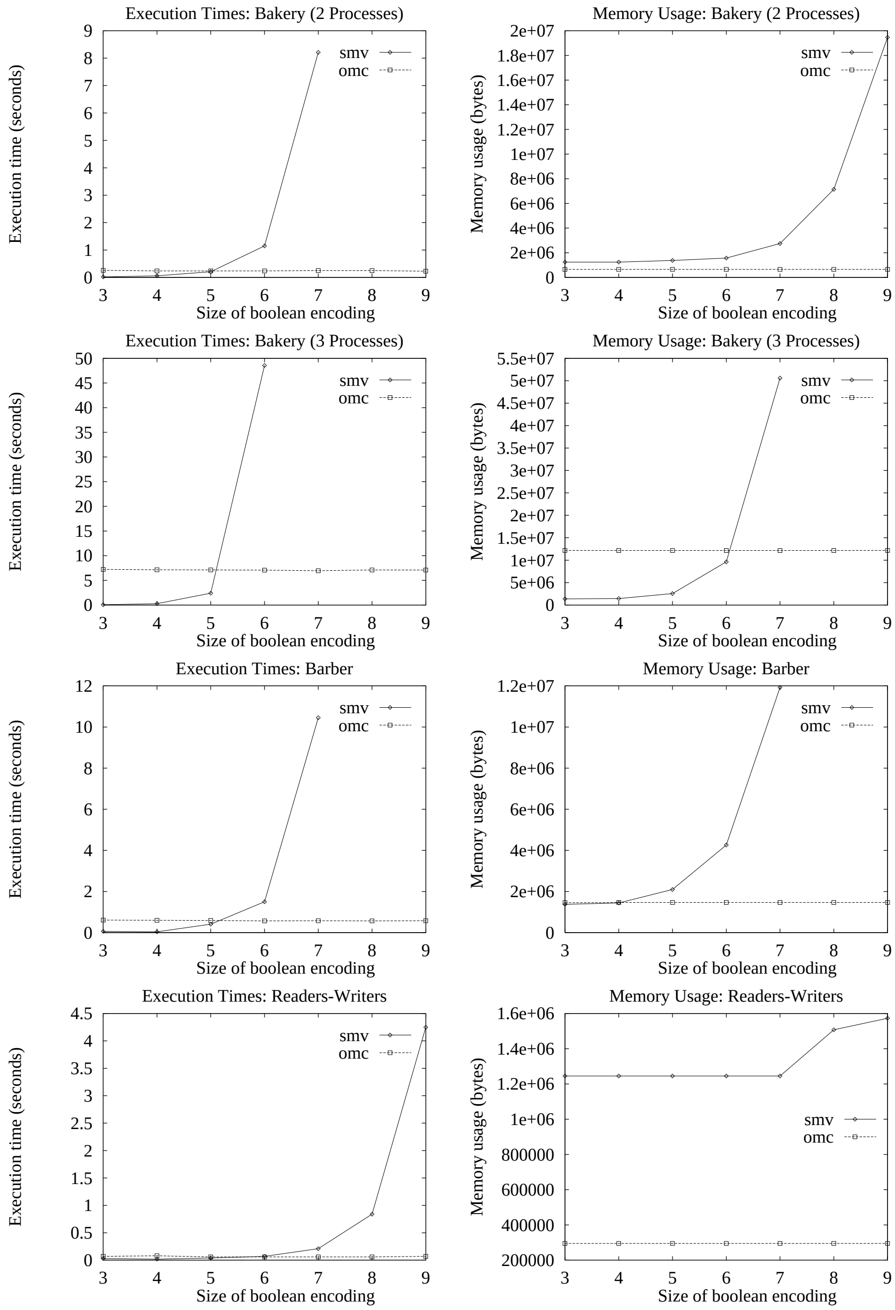

Fig. 1. Execution times and memory usage for SMV and OMC in verification of BAKERY, BARBER and READERS-WRITERS

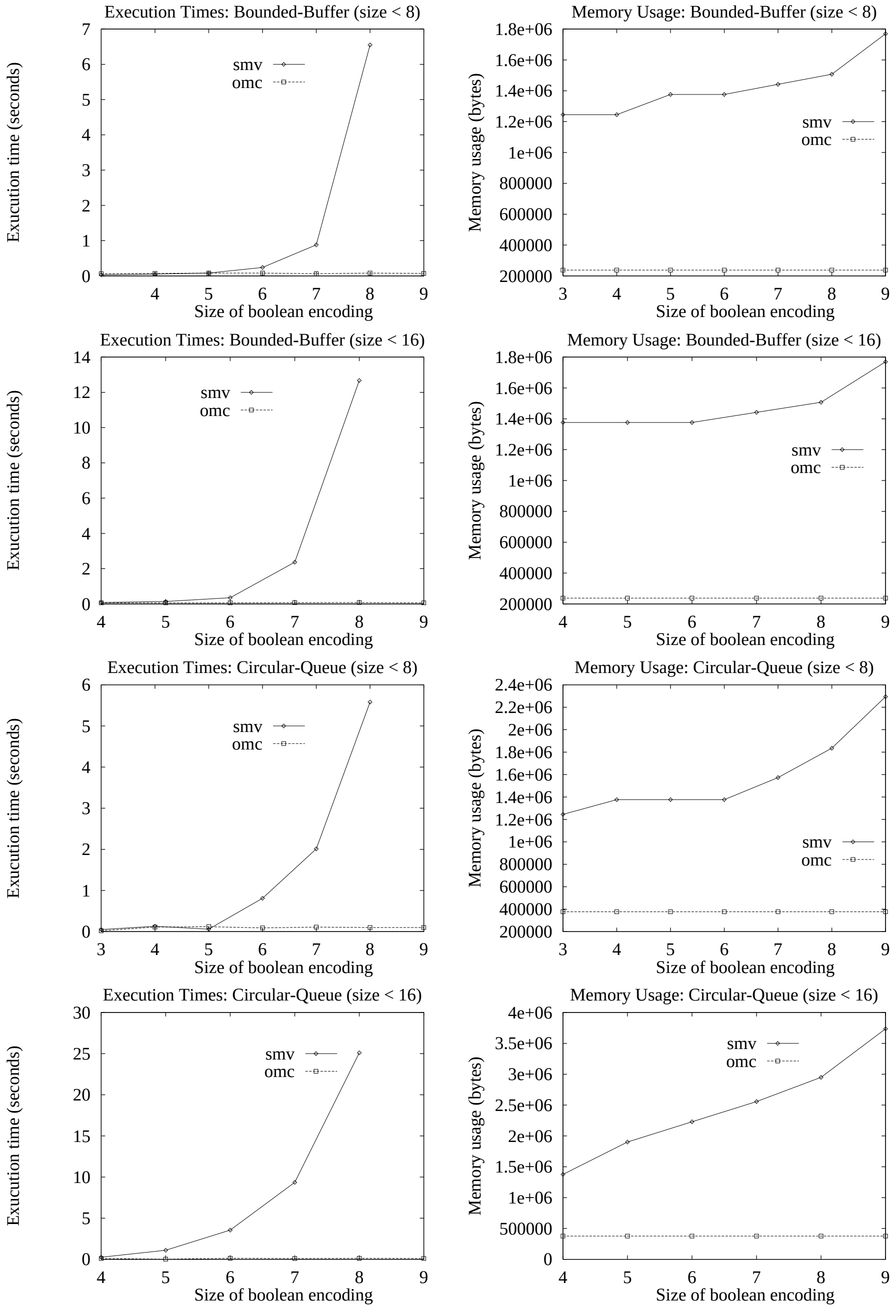

Fig. 2. Execution times and memory usage for SMV and OMC in verification of BOUNDED-BUFFER and CIRCULAR-QUEUE

Table 2. Experiments where each integer variable i is restricted to $0 \le i < 32$. In bounded-buffer and circular-queue instances the parameterized constant $size$ is restricted to $0 \le size < 16$ ($\uparrow$ denotes that the fixpoint computations did not converge)

Problem Instance	SMV Execution Time (seconds)	SMV Memory Usage (Kbytes)	OMC Execution Time (seconds)	OMC Memory Usage (Kbytes)
BAKERY (2 processes)	0.20	1376	0.27	655
BAKERY (3 processes)	2.45	2556	7.28	12165
BAKERY (4 processes)	30.38	4719	$\uparrow$	$\uparrow$
BARBER	0.43	2163	0.56	1458
BARBER-1	0.43	2163	$\uparrow$	$\uparrow$
BARBER-2	0.38	1966	0.28	926
BARBER-3	0.44	2163	0.16	655
READERS-WRITERS	0.03	1245	0.05	295
BOUNDED-BUFFER	0.14	1376	0.06	238
BOUNDED-BUFFER-1	0.15	1376	0.06	188
BOUNDED-BUFFER-2	0.14	1376	0.05	147
BOUNDED-BUFFER-3	0.15	1376	$\uparrow$	$\uparrow$
CIRCULAR-QUEUE	1.10	1901	0.11	377
CIRCULAR-QUEUE-1	1.12	1901	$\uparrow$	$\uparrow$
CIRCULAR-QUEUE-2	1.29	1901	0.09	328

bedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, March 1996.

[And91] G. R. Andrews. *Concurrent Programming: Principles and Practice*. The Benjamin/Cummings Publishing Company, Redwood City, California, 1991.

[BCL91] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In *Proceedings of the International Conference on Very Large Scale Integration*, August 1991.

[BCM+90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. H. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, January 1990.

[BGL98] T. Bultan, R. Gerber, and C. League. Verifying systems with integer constraints and boolean predicates: A composite approach. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 113–123, March 1998.

[BGP] T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: Symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems*, to appear.

[BGP97] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using Presburger arithmetic. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 400–411. Springer, June 1997.

[BKR96] M. Biehl, N. Klarlund, and T. Rauhe. Mona: Decidable arithmetic in practice. In *Proceedings of Formal Techniques in Real-Time and Fault-Tolerant Systems, 4th International Symposium*, volume 1135 of *Lecture Notes in Computer Science*. Springer, 1996.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[CAB+98] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.

[CABN97] W. Chan, R. J. Anderson, P. Beame, and D. Notkin. Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 316–327. Springer, June 1997.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CGL92] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 343–354, 1992.

[CZ95] E. Clarke and X. Zhao. Word level symbolic model checking: A new approach for verifying arithmetic circuits. Technical Report CMU-CS-95-161, School of Computer Science, Carnegie Mellon University, May 1995.

[DP99] G. Delzanno and A. Podelski. Model checking in CLP. In Rance Cleaveland, editor, *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 223–239. Springer, March 1999.

[HKL+98] C. L. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on Software Engineering*, 24(11):927–948, November 1998.

[KMM+97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 400–411. Springer, June 1997.

[KMP+95] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega library interface guide. Technical Report CS-TR-3445, Department of Computer Science, University of Maryland, College Park, March 1995.

[KSA98] J. H. Kukula, T. R. Shiple, and A. Aziz. Implicit state enumeration for FSMs with datapaths. In *Proceedings of Formal Methods in Computer-Aided Design*, 1998.

[McM93] K. L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, Massachusetts, 1993.

[SKR98] T. R. Shiple, J. H. Kukula, and R. K. Ranjan. A comparison of Presburger engines for EFSM reachability. In *Proceedings of the 10th International Conference on Computer-Aided Verification*, 1998.

[WB95] P. Wolper and B. Boigelot. An automata-theoretic approach to Presburger arithmetic constraints. In *Proceedings of the Static Analysis Symposium*, September 1995.