Parallel Program Verification Using BMC

Toni Jussila, Ilkka Niemelä[1][2]

Abstract. The paper describes a research tool for studying the applicability of bounded model checking techniques to verifying asynchronous software. A simple programming language, PROMELAb, is introduced and an operational semantics for it is defined. The language is basically a subset of PROMELA, the input language of the SPIN model checker. The basic idea is to study techniques for improving the efficiency of bounded model checking techniques for parallel software by exploiting the potential concurrency in executing asynchronous programs. As a first step, a translation from a PROMELAb program and an execution bound to a propositional formula is devised such that the models of the formula capture exactly the bounded executions of the program. The translation allows the analysis of both interleaving and step executions. It is shown how it can be used to check bounded reachability properties and how the approach extends to handling safety properties. Techniques for optimizing the encoding are also discussed.

1 INTRODUCTION

Model checking is a verification technique for the analysis of reactive systems. Capable of finding subtle errors in real commercial designs and being largely automatic it is gaining wide industrial acceptance. However, the technique has to handle the fact that the state spaces of complex system models may grow intractably large.

In symbolic model checking the state space is represented compactly by encoding it as a boolean decision diagram (BDD), a canonical form for boolean expressions. Model checkers using BDDs are usually capable of handling hundreds of state variables. However, the BDDs may also grow infeasibly large and, in addition, their size depends heavily on a correct variable ordering.

In 1998, Biere et. al presented a symbolic model checking technique using SAT procedures instead of BDDs [4]. The basic idea is to consider counterexamples of a particular length k and generate a propositional formula that is satisfiable iff a counterexample of such length exists. This parameter is the justification for the coined name bounded model checking.

Bounded model checking techniques have been applied quite successfully to verification of (synchronous) hardware designs, see, e.g. [5]. Recently, bounded model checking methods for asynchronous systems have been studied using, e.g. Petri nets [7, 6] and PROMELA [2] as the system description languages.

[1] Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory of Theoretical Computer Science
P.O. Box 5400
02015 HUT, Finland

[2] The work of the first author has been supported by the Academy of Finland (Project 53695), the Helsinki Graduate School in Computer Science and Engineering (HeCSE) and Nokia Foundation and the second author by the Academy of Finland (Project 53695).

This paper addresses also asynchronous systems and describes ongoing work on applying bounded model checking techniques to verifying parallel programs in the area of protocol and controller software. As the target programming language a subset of PROMELA called PROMELAb has been chosen. The basic idea is to develop a research tool for studying novel techniques for improving the efficiency of bounded model checking techniques for parallel software by exploiting the potential concurrency in executing asynchronous programs. As the first step we present a translation from a PROMELAb program and a execution bound to a propositional formula which captures in a compact form the bounded executions of the program. The encoding is designed so that standard interleaving execution as well as step executions (allowing concurrent actions) can be captured by adding suitable conditions. Step executions are interesting from the efficiency point of view because using steps a larger number of reachable states can be covered than with interleaving executions using the same bound for the length of the execution. Moreover, a number of optimization techniques which enhance the compactness of the encoding are identified.

The approach presented is related to the tool `p2b` [2] which also enables the use of symbolic model checking techniques for PROMELA programs. This is done by translating a PROMELA program to the input language of SMV and then using the BDD-based model checking techniques available in SMV. The goal here is to study SAT-based bounded model checking for PROMELA type languages and for this a direct translation to propositional logic is developed. The idea is to enhance the efficiency and range of applicability of bounded model checking by investigating optimization techniques that directly exploit information that can be extracted from the source code of the program.

Section 2 introduces the programming language PROMELAb and Section 3 gives its operational semantics. In Section 4, the translation of valid PROMELAb programs to boolean formulae is presented, followed by the encoding of the reachability and safety properties (Section 5). Section 6 discusses optimizations available in the implementation with some initial experimental results and Section 7 concludes the paper.

2 PROGRAMMING LANGUAGE PROMELAb

The target domain for the presented work is model checking control and protocol software. For this purpose a simple programming language PROMELAb has been devised. The language borrows a lot from PROMELA, the input language of the SPIN system [8], actually being a subset.

A typical PROMELAb program consists of parallel components capable of communicating through global variables and FIFO queues. The central issues in the design of the language is, on the one hand, the provision of an intuitive formalism for parallel programming. On the other hand, it should support the constructs often needed in formal verification that may be different from traditional programming languages.

PROMELAb supports integer variables, typically used with a smaller value range than in traditional computer architecture [3]. This is justified with the assumption that in the target domain, protocol software, the models normally contain integer counters not assuming very large values. The values of variables may be modified with statements like `i = j + 2;`.

The communication through FIFO queues is implemented with the constructs taken from PROMELA. For instance, the declaration

$$\texttt{chan i} = [2]\ \texttt{of}\ \{\texttt{int},\ \texttt{int}\};$$

creates a channel named `i` having two slots and the message carried consists of two integers. A message may be sent with a command like `i!1,2;` and read (to variables `a` and `b`) with the command `i?a,b;`.

A useful programming language has to provide means for branching. PROMELAb adopts the unconditional and conditional structures from PROMELA. The statement `goto la;` generates a jump to the label `la` and the structure from Figure 1 models a nondeterministic branching, in which the execution proceeds from the branch of one of the conditions (represented by `ca,...,cn`) evaluating to true.

$$\begin{array}{l}
\texttt{l0} : \texttt{if} \\
\qquad :: (\texttt{ca}) \rightarrow \texttt{la} : \ldots \\
\qquad \vdots \\
\qquad :: (\texttt{cn}) \rightarrow \texttt{ln} : \ldots \\
\quad \texttt{fi};
\end{array}$$

Figure 1. Nondeterministic if

To facilitate the use of control flow constructs similar to those of C and JAVA too, it is possible to write a statement like `if(c1){ ...} else if (c2)...` having the deterministic semantics. In addition, a `while` block is possible. Both of these are syntactic sugar representable using the constructs defined above.

Compared to C, PROMELAb makes the restriction of not allowing side effects in the boolean conditions used in testing for branching. Thus, a statement like `if (a = 2)...` is not a valid one. Furthermore, the results of boolean conditions are regarded to be of different type than the results of arithmetic expressions and therefore the outermost operator in the condition has to have a boolean result (for instance, `if (1 + 2)...` is not a valid statement).

3 OPERATIONAL SEMANTICS

A complete PROMELAb program consists of n parallel components together with a SPIN-style run block, in which they are initialized. Figure 2 contains the PROMELAb model of a simple producer-consumer problem, where the component p sends a message that the component q receives.

[3] The current implementation has the width as a compilation option.

The execution of a PROMELAb program is modelled with an operational semantics similar to [11] considering executions to consists of a sequence of *program states*. A program state is a $n + 2$ tuple

$$(\mu, \lambda_1, \ldots, \lambda_n, \kappa) \tag{1}$$

where μ is the memory state (the values of the variables and the queue contents), λ_i is the control flow position within each component and κ is the scheduling component giving the component from which the next statement to be executed is chosen.

```
chan i = [1] = {int};

p() {
  i!1;
}

q() {
  int a;
  i?a;
}

init() {
  run p();
  run q();
}
```

Figure 2. PROMELAb model of the producer-consumer problem.

Starting from the initial state, the next state in each step is obviously determined by the statement executed, given by κ and the statement in the chosen component. Firstly, the control flow positions in all the other components remain unchanged. In the active component, the control position is determined by the statement type. An unconditional jump causes it to move to that label, an assignment moves it to the next statement etc. If the executed statement is a jump (conditional or unconditional), the memory state remains the same, assignments and queue operations may cause it to change.

A component may not be schedulable if its current statement is a queue send or receive operation to a full or empty queue, respectively. Furthermore, this may be the case when it has successfully terminated.

An *execution* is either an infinite sequence of programs states or a finite one. The latter means that all the components have successfully terminated or there is a deadlock, distinguished by the fact that at least one component has not finished. The semantics is maximal in the sense that a finite sequence of programs states where some component could be scheduled in the last state, is not considered an execution.

4 TRANSLATION

The encoding of valid programs follows the ideas of SAT planning [10]. Even though the goal is a propositional formula, the ideas are best represented using formulae with predicates. The predicates used in the translation of PROMELAb programs together with their descriptions are given in Table 1.

Given a PROMELAb program P and a bound k, the translation produces a formula Ψ_P^k which is structured into two parts $\Pi_P^k \wedge \Sigma_P^k$.

Formula Π_P^k captures the conditions on possible executions of P up to k in such a way that the standard interleaving executions as well as step executions can be obtained by providing suitable constraints on scheduling the components using the formula Σ_P^k. First, the encoding of executions Π_P^k is presented and then it is demonstrated how interleaving and step executions can be captured using suitable formulae Σ_P^k.

The basic idea of the encoding Π_P^k is illustrated in Figure 3. Atoms $a_j(t)$ for a binary encoding of the memory state are introduced for each time point $t = 0, 1, \ldots, k$. For every label l in each component in P, there is an atom $cf(l, t)$ for every $t = 0, 1, \ldots, k$. Similarly, there is an atom $qu(q, c, t)$ for each queue q and its possible usage count $c = 0, 1, \ldots, qmax$ where $qmax$ is the capacity of q. Finally, there is an atom $sc(p, t)$ for each component p in P for every $t = 0, 1, \ldots, k-1$.

Then formulae (α) that encode the change from state t to $t+1$ and, on the other hand, formulae constraining the state k (β) according to the statements in program P are provided.

Table 1. Translation Predicates

Predicate	Description
$sc(p,t)$	Component p is scheduled at time t.
$cf(l,t)$	Control flow is at label l at time t.
$qu(q,c,t)$	c slots of queue q are used at time t.

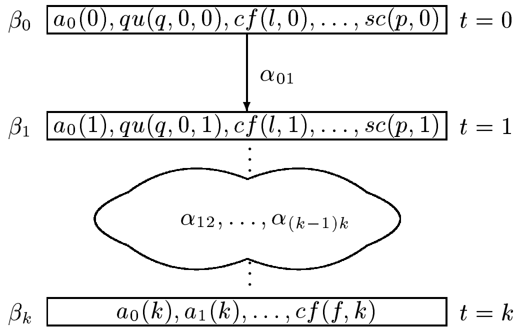

Figure 3. State vectors and formulae

The encoding of Π_P^k involves two rather orthogonal issues. Firstly, there is the issue of representing the memory state. This involves also the encoding of boolean conditions and arithmetic expressions operating on variable values. Secondly, there is the representation of the control flow.

In the following, first the encoding of the boolean conditions and the arithmetic expressions is explained. In Section 4.3 the encoding of statements is presented. Sections 4.5 and 4.6 describe the differences in the encoding of Σ_P^k in the interleaving and step execution models and the last section estimates the size of the resulting formula.

4.1 Boolean Conditions

To correctly encode a boolean condition, a formula having a model iff the condition in question holds has to be presented. It turns out that it is sufficient to present explicit encodings for only three cases, the others follow by either negating the condition or switching the order of arguments. In the formulae below, the variables are represented using a binary encoding. The bits of the variables run from one to n (the most significant bit, MSB) and the term l_i is used to refer to the bit i of the variable l.

4.1.1 Equality

The simplest boolean condition is equality. Obviously, the values of two variables are equal, iff all their bits are equivalent. Thus, equality is encoded with the following formula:

$$l == r \equiv_{def} \bigwedge_{1 \le i \le n} (l_i \leftrightarrow r_i) \tag{2}$$

4.1.2 Less than (with equality)

The encoding of an expression of the type $l < r$ follows the following intuition. Starting from the MSB, the condition holds, if the bits in the left and right arguments are zero and one, respectively. Otherwise, if they are equal, the comparison has to be extended to the next bit. The intuition is caught in the following formula:

$$\begin{aligned} l < r \equiv_{def} & (L_n \leftrightarrow (\neg l_n \wedge r_n)) \wedge \\ & (E_n \leftrightarrow (l_n \leftrightarrow r_n)) \wedge \\ & \bigwedge_{1 \le i < n} (L_{n-i} \leftrightarrow (E_{n-i+1} \wedge (\neg l_{n-i} \wedge r_{n-i}))) \wedge \\ & \bigwedge_{1 \le i < n} (E_{n-i} \leftrightarrow (E_{n-i+1} \wedge (l_{n-i} \leftrightarrow r_{n-i}))) \wedge \\ & \bigvee_{1 \le i < n} L_i \end{aligned} \tag{3}$$

To encode the case with equality ($l \le r$) only the following modification has to be made:

$$L_1 \leftrightarrow (E_2 \wedge (\neg l_1 \vee r_1)) \tag{4}$$

4.1.3 Other Cases

Other boolean conditions can be obtained from the cases above with the conversions given in Table (2).

Table 2. Other boolean conditions

$l \neq r$	$\equiv$	$\neg(l = r)$
$l > r$	$\equiv$	$r < l$
$l \geq r$	$\equiv$	$r \leq l$

4.2 Arithmetic Expressions

PROMELAb supports the standard mathematical operations of addition, subtraction, multiplication, integer division and modulo. The difference between them and the boolean conditions is that instead of merely having a truth value they have a result. In the formulae below the result of an operation can be obtained from the values of the bit vector $x_1, \ldots, x_n$, corresponding to the operation $x = l\ op\ r$. It should be noted that several designs exist for implementing the arithmetic operations. The ones presented are chosen due to their simplicity.

4.2.1 Addition

Addition is a binary version of the elementary school algorithm with each bit possible toggling a carry bit to be set (carry bits denoted c_i).

$$
\begin{aligned}
x = l + r \equiv_{def} & (x_1 \leftrightarrow (l_1 \oplus r_1)) \wedge \\
& (c_1 \leftrightarrow (l_1 \wedge r_1) \wedge \\
& \bigwedge_{1<i\leq n} (x_1 \leftrightarrow (l_i \oplus r_i \oplus c_{i-1})) \wedge \\
& \bigwedge_{1<i<n} (c_i \leftrightarrow ((l_i \wedge r_i) \vee (c_{i-1} \wedge (l_i \vee r_i))))
\end{aligned} \tag{5}
$$

4.2.2 Subtraction

Subtraction can be encoded as follows:

$$
\begin{aligned}
x = l - r \equiv_{def} & (x_1 \leftrightarrow (l_1 \oplus r_1)) \wedge \\
& (c_1 \leftrightarrow (\neg l_1 \wedge r_1) \wedge \\
& \bigwedge_{1<i\leq n} (x_1 \leftrightarrow (l_i \oplus r_i \oplus c_{i-1})) \wedge \\
& \bigwedge_{1<i<n} (c_i \leftrightarrow ((l_i \wedge r_i \wedge c_{i-1}) \vee (\neg l_i \wedge (r_i \vee c_{i-1}))))
\end{aligned} \tag{6}
$$

4.2.3 Multiplication

The encoding of multiplication also follows the tradition of high school algebra, i.e. the left argument of the operation is multiplied by each bit of the right argument and result is shifted left by the amount defined by its bit position. These results are then appropriately added. A single bit multiplication is simply just the binary and operation. Since addition is already presented it is omitted from the formula above and denoted with $enc(x = a + b)$, a convention also applied later. In addition, the shorthand T_i is used to refer to the bit vector $T_{ni}, \ldots, T_{1i}$ and the term $(T_i << j)$ refers to shifting the bits of T_i to the left j steps.

$$
\begin{aligned}
x = l * r \equiv_{def} & \bigwedge_{1\leq i,j\leq n} (T_{ij} \leftrightarrow l_i \wedge r_i) \wedge \\
& enc(x = \Sigma_{1\leq i\leq n}(T_i << (i-1)))
\end{aligned} \tag{7}
$$

4.2.4 Division and Modulo

The division and modulo algorithms can be encoded with the following formula:

$$
\begin{aligned}
x = l/r \equiv_{def} & enc(R_n = l) \wedge enc(D_n = (r << (n-1))) \\
& \bigwedge_{1\leq i\leq n} (x_i \leftrightarrow enc(D_i \leq R_i)) \\
& \bigwedge_{1<i\leq n} ((x_i \rightarrow enc(R_{n-1} = R_n - D_n)) \wedge \\
& (\neg x_i \rightarrow enc(R_{n-1} = R_n)) \wedge \\
& enc(D_{n-1} = D_n >> 1))
\end{aligned} \tag{8}
$$

The difference with the formula above when computing the remainder is that instead of setting the result bits according to the comparision result $D_i \leq R_i$, the result is obtained by setting the value of x equivalent to R_1.

4.3 Statements

This section presents the encoding of the different statement types. The notational shorthand applied above is augmented with the time step. Thus, when describing that x has the value $expr$ at time t the abbreviation $enc(x(t) = expr)$ is used. Furthermore, for clarity it is assumed that each statement carries a unique label (typically `la:` is used).

Firstly, let

$$ex(la, t) \leftrightarrow (sc(p, t) \wedge cf(la, t)) \tag{9}$$

define the predicate ex meaning that a statement is executed. This is the case iff the component it is in is scheduled and the control flow is in that position.

4.3.1 Assignments

An assignment is a statement of the form `la: x = expr; lb: ...`, `lb` being the label of the next statement. The formula it generates is an implication having as the precondition the requirement that the statement is executed. Assuming the predicate ex as above, an assignment is modelled with the following formula:

$$ex(la, t) \rightarrow (enc(x(t+1) = expr) \wedge cf(lb, t+1)) \tag{10}$$

4.3.2 Unconditional Jumps

An unconditional jump is a statement of the form `la: goto ln;`. The encoding is straight forward:

$$ex(la, t) \rightarrow cf(ln, t+1) \tag{11}$$

4.3.3 Conditional Branching

Recall the structure from Figure 1. It represents a nondeterministic branching based on the truth value of the conditions. Assuming there to be n branches, with the conditions and the labels as in the figure and the statement after the entire structure to be $l1$, the encoding is:

$$
\begin{aligned}
ex(l0, t) \rightarrow & (cf(la, t+1) \vee \cdots \vee cf(ln, t+1) \\
& \vee cf(l1, t+1))
\end{aligned} \tag{12}
$$

$$cf(la, t+1) \to enc(c1) \tag{13}$$

$$\vdots$$

$$cf(ln, t+1) \to enc(cn) \tag{14}$$

$$cf(l1, t+1) \to (\neg enc(c1) \wedge \cdots \wedge \neg enc(cn)) \tag{15}$$

4.3.4 *Queue Operations*

The encoding of the queue operations are the most complex, since they need to maintain the control flow, the queue contents and the queue usage parameters correctly. Consider first the statement `la: q!m; lb:` sending the message `m` to the queue `q`. Executing this should cause the message to be copied to the first available queue slot (guided by the qu-predicate), the number of used queue slots to be incremented and the control flow to be advanced to the next statement. Assuming the notational convention of q_i to refer to the queue slot i and the queue slots to start from one, the statement effects are modelled with the following formulae:

$$ex(la, t) \to \neg qu(q, qc, t) \tag{16}$$

$$ex(la, t) \to cf(lb, t+1) \tag{17}$$

$$ex(la, t) \to \bigwedge_{0 \leq i < qc} (qu(q, i, t) \to enc(q_{(i+1)}(t+1) = x) \wedge qu(q, i+1, t+1)) \tag{18}$$

$$ex(la, t) \to \bigwedge_{1 \leq i \leq qc} (\neg qu(q, i, t+1) \to (q_i(t+1) \leftrightarrow q_i(t))) \tag{19}$$

In the first formula the meaning of the term $\neg qu(q, qc, t)$ is to disallow queue send commands to full queues. The purpose of the last formula is to maintain the contents of the other queue slots unaltered. Consider next the statement `la:q?a; lb:`, reading from the queue to the variable a. Such a statement is not allowed if the queue is empty. Its execution causes the contents of the first queue slot to be copied to the variable, the number of used queue slots to be decremented and the contents of the other queue slots to be shifted down.

$$ex(la, t) \to \neg qu(q, 0, t) \tag{20}$$

$$ex(la, t) \to cf(lb, t+1) \tag{21}$$

$$ex(la, t) \to enc(x(t+1) = q_{1t}) \tag{22}$$

$$ex(la, t) \to \bigwedge_{1 \leq i \leq qc} (qu(q, i, t) \to qu(q, i-1, t+1)) \tag{23}$$

$$ex(la, t) \to \bigwedge_{0 \leq i < qc} (enc(q_i(t+1) = q_{(i+1)}(t))) \tag{24}$$

4.3.5 *Frame Conditions*

When planning problems are translated to SAT and only the effects of the operations are modelled, the resulting formula may have many unintended models. In the present domain, the problem would be manifested for instance in the case of an assignment not being executed. Then the implication modelling its effect would be true by its precondition being false and any combination of the bits for the value of the variable would still constitute a valid model. This is known as the frame problem [12] and this section presents the necessary frame axioms to disallow the unintended models.

The problem of ruling out models allowing variables to assume undesired values is taken care of with a formula demanding that a statement modifying the variable is executed or it retains its value from the previous time step.

$$(sc(p, t) \wedge (cf(la, t) \vee \cdots \vee cf(ln, t))) \vee enc(a_{t+1} = a_t) \tag{25}$$

The disjunction $(cf(la, t) \vee \ldots \vee cf(ln, t))$ must contain all the statements capable of modifying the variable at time t. It should also be noted that if the variable is global, all the components having statements modifying it have to be part of the formula. A similar issue is that of control flow. If a component is not chosen for execution, the control flow has to stay in the same position.

$$sc(p, t) \vee ((cf(la, t+1) \leftrightarrow cf(la, t)) \wedge \cdots \wedge (cf(ln, t+1) \leftrightarrow cf(ln, t))) \tag{26}$$

The conjunction must contain all the statements in the component that in some execution may be executed at time t. However, the above formula is not enough, since the control flow may still be "split" (several statements within the component to be executable) in the case of the component being active. The following formula guarantees the uniqueness of the control flow:

$$[1, 1]\{cf(la, t), \ldots, cf(ln, t)\} \tag{27}$$

The formula above is a convenient shorthand. The numbers in the angular brackets give the lower and upper bounds on allowed true literals in the following set. Notice, as in this case, that both may be the same. This may be presented in propositional logic by first disjunctively demanding that at least one of them is true and then ruling out pairs.

Finally is the maintenance of the FIFO queues. If a queue manipulating statement is not executed both the contents and the qu-predicate have to be maintained.

$$(sc(p, t) \wedge (cf(la, t) \vee \cdots \vee cf(ln, t))) \vee (\bigwedge_{1 \leq i \leq qc} (enc(q_i(t+1) = q_i(t))) \wedge (\bigwedge_{0 \leq i \leq qc} (qu(q, i, t+1) \leftrightarrow qu(q, i, t))) \tag{28}$$

To be consistent, the qu-predicate has to be unique for all the queues in all time steps. This is guaranteed by the following formula:

$$[1, 1]\{qu(q, 0, t), \ldots, qu(q, qc, t)\} \tag{29}$$

4.4 Initial State

When a parallel program starts its execution the control flow positions and the memory state is known. This has to be reflected in the encoding by setting the corresponding cf-predicates to true and setting the bits of the variables according to the initial memory state. The uninitialized variables may be implicitly assumed to be zero or to have any value, by negating the bits or leaving them undefined, respectively.

4.5 Interleaving Semantics

Interleaving semantics is a model in which in each time step, precisely one component is scheduled. To guarantee this in the models of the generated formula Ψ_P^k, the following formula Σ_P^k is needed (assuming the components of the parallel program to be $p_1, \ldots, p_n$:

$$[1,1]\{sc(p_1,t), \ldots, sc(p_n,t)\} \tag{30}$$

for $t = 0, \ldots, k-1$.

The following theorem establishes a one-to-one correspondence between the presented encoding and the prefixes of the executions in the operational semantics.

Theorem 1 *The encoding with the interleaving semantics is sound and complete with respect to the operational semantics, i.e. each model of the formula obtained by translating a PROMELAb program with the parameter k corresponds to a prefix of an execution of length k and vice versa.*

Proof (sketch) Consider a mapping relating the memory state of the operational semantics to the binary encoding of the variables, the control flow positions in each time step to the cf-predicate and the scheduling component κ to the sc-predicate.

The proof is based on an inductive argument over the program states. Soundness (every model corresponds to a prefix of an executions sequence) is proven by considering a model $\mathcal{M}$ and from the conditions it has to fulfill it follows that with the mapping above, the obtained execution sequence is a valid one. Completeness (every prefix of execution sequences has a corresponding model) is proven by considering an arbitrary prefix, mapping it as above and proving that the obtained demands constitute a model of the formula.

4.6 Step Semantics

The basic idea in the step semantics is to allow the simultaneous execution of several statements from different components as long as no reachable state is lost. This is achieved by requiring that the statements in a step are independent in the sense that executing them in any order leads to the same state. In this paper a straightforward technique for this is studied. Two statements are considered dependent if they have mutual variables. The variables may appear in the left or right hand side in assignments, in queue operations or in boolean conditions. Statements which do not have mutual variables are considered independent. Hence, a step preserves all reachable states if it does not contain a dependent pair of statements (labels). Following these observations the encoding Σ_P^k allowing step executions consists of the formulae

$$sc(p_1,t) \vee \cdots \vee sc(p_n,t) \tag{31}$$

$$[0,1]\{ex(l1,t), \ldots, ex(ln,t)\} \tag{32}$$

for every $t = 0, 1, \ldots, k-1$ and formula (32) consists of statements having a particular mutual variable.

4.7 Compactness

An important measure in assessing the benefits of the encoding is its size. This section discusses the size of the generated formula obtained from the translation of a PROMELAb program. Firstly, the entire formula is a conjunction of the formulae presented above. The conjuncts modelling the effects of the statements have to be encoded for all the time steps in which their execution is possible in some execution. Thus, for each statement, their effects have to appear $\mathcal{O}(k)$ times. The frame conditions obviously need to apply for all the time steps as well as the restrictions guaranteeing interleaving (or step) semantics. Formula (25) has to be constructed for each variable and formula (26) for each component. Similarly, formula (28) has to appear for each queue. From these, the following lemma can be established:

Lemma 1 *The encoding of a PROMELAb program with the parameter k is a conjunction of $\mathcal{O}((s+v+p+q)*k)$ formulae, where s is the number of statements, v is the number of variables, p is the number of parallel components and q is the number of queues.*

5 SPECIFYING PROPERTIES

The properties used in verification are usually expressed using temporal logic. The standard classification of verification goals divides properties into four categories [3]:

- *Reachability properties* state that some particular state can be reached.
- *Safety properties* express that under some conditions something never happens
- *Liveness properties* demand that under some conditions something will ultimately occur.
- *Fairness properties* state that under some conditions something will (or will not) occur infinitely often.

The present work considers only the first category and discusses a technique with which the second class, safety, can be expressed using reachability. Recall that the translation of a PROMELAb program P with bound k resulted in a formula Ψ_P^k having as its models prefixes of operational execution sequences. Given a property q, the model checking procedure is to give a witness if a state in which q holds can be reached within k steps. It is straightforward to satisfy this demand. Firstly, the property q, is encoded as follows ($q(i)$ refers to the fact that q holds at time t):

$$\phi = q(0) \vee \cdots \vee q(k) \tag{33}$$

Then, the formula $\Psi_p^k \wedge \phi$ has a model iff the program p has an execution of length k in which q holds in some state.

It is fairly obvious that a negative reachability properties is actually safety property where the concept "under certain conditions" is missing. In general, one elegant way of characterizing safety properties is a syntactic one, namely, a safety property is of the form $\mathbf{G}\phi^-$, where ϕ^- is a *past* temporal formula. One example of this type of formula is the following:

$$\mathbf{G}(st \rightarrow \mathbf{F}^{-1}key) \tag{34}$$

This can be given the meaning that if the car starts (st true) then there was some earlier moment when the key was inserted. The verification of any property of this type can be reduced to verifying a reachability property. The technique is to introduce so called *history variables* [3] being true in a state iff the given past formula holds in that state.

6 IMPLEMENTATION

Currently, an initial implementation [4] of the presented ideas exists. It follows the structure illustrated in Figure 4.

[4] available at `http://www.tcs.hut.fi/~tjussila/promelab`

A compiler, called `p2bc` reads PROMELAb source and translates it to a boolean representation. The resulting encoding can then either be solved directly with a tool called BCSat [9] or translated to a CNF representation [5] and analyzed with standard SAT solvers. In the translation, general propositional formulae (instead of conjunctive normal form) are used as the target language to have more flexibility and expressiveness to capture in a compact form complicated interactions caused, e.g., by frame axioms and queue behavior. For representing such general formulae boolean circuits [9] are used because they enhance compactness by allowing structure-sharing. If needed, boolean circuits can be translated to a compact CNF representation by introducing new atoms for gates [9].

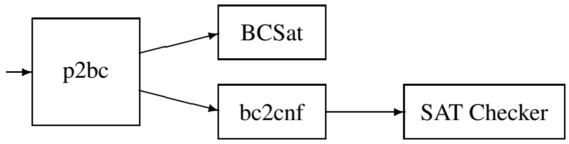

Figure 4. Tool chain in the verification procedure

It turns out that the actual encoding may benefit from some techniques familiar from compiler construction, most notably the following:

- Control flow graph:
  This is a rather straightforward optimization relating to the earliest moment a statement may be executable. This involves the implicit construction of the control flow graph [1] of the program and inferring the earliest moment for a statement from that of its immediate predecessors. The last possible moment depends on the other components and this technique does not help.
- Goto inline:
  One straight forward optimization is to avoid the labels of goto statements by having the statements reaching a goto to proceed straight to the target of the unconditional jump.
- While optimization:
  Above it was stated that a `while`-statement is syntactic sugar expressible with the nondeterministic if and an unconditional jump. Obviously, goto inlining handles the elimination of the otherwise extra time step.
- Constant folding:
  It should be noted that the boolean conditions and the arithmetic expressions are presented above in the most general case, i.e. between two unknown variables. The formulae are remarkably simplified if the other argument is a constant. For instance, a division with a power of two reduces to a trivial shift operation compared to the formula above.
  The situation is even simpler, if the value of both of them can be staticly inferred from the program text. Then, the condition or expression can be evaluated and thus the corresponding subformula can be omitted. This technique, known as constant folding is applied in the implementation as well.

Tables 3 and 4 present initial results of two examples, a dining philosophers problem and a simple mutex protocol. In the first example, each philosopher tries first to take its left fork and then the right and the system deadlocks if each philosopher manages to take his left fork. The philosophers are modelled as processes and the forks are channels with a capacity of one, each channel shared by two processes. The task is to verify the system for deadlock.

The mutex protocol consists of n processes that communicate with an arbiter via channels a_i, b_i and c, $1 \leq i \leq n$, where each channel is of capacity one. A process P_i sends its request for entering a critical section to the arbiter via a_i; the arbiter elects one such process, say P_j and sends it a grant for entering the critical section via b_j. That process sends a notification via c to the arbiter once it has left the critical section [2]. The task is to verify that the system guarantees mutual exclusion.

It turns out that the performance with the interleaving model compares favorably to SPIN and SMV run times [2] whereas even better results are obtained with the step semantics.

Table 3. Initial results - dining philosophers

#phil.	in$_{bc}$	in$_{ch}$	step$_{bc}$	step$_{ch}$
5	0.11/5	0.04/5	0.01/1	0.01/1
6	0.23/6	0.04/6	0.01/1	0.01/1
7	0.54/7	0.07/7	0.01/1	0.01/1
8	1.08/8	0.13/8	0.01/1	0.01/1
9	2.09/9	0.21/9	0.01/1	0.01/1
10	3.66/10	35.59/10	0.01/1	0.01/1
11	6.20/11	87.09/11	0.01/1	0.01/1
12	38.73/12	2716.00/12	0.01/1	0.01/1
13	602.19/13	$>$ 1h/13	0.01/1	0.01/1
14	$>$ 1h/14	$>$ 1h/14	0.01/1	0.01/1

Table 4. Initial results - mutex protocol

#proc.	in$_{bc}$	in$_{ch}$	step$_{bc}$	step$_{ch}$
3	0.21/8	0.09/8	0.07/6	0.04/6
4	0.77/9	0.20/9	0.08/6	0.06/6
5	2.33/10	0.50/10	0.15/6	0.09/6
6	5.95/11	2.00/11	0.25/6	0.17/6
7	17.15/12	2.03/12	0.32/6	0.18/6
8	71.81/13	25.89/13	0.32/6	0.23/6
9	791.69/14	109.22/14	1.37/6	0.30/6
10	$>$1h/15	122.07/15	1.81/6	0.38/6
11	$>$1h/16	262.84/16	2.14/6	0.54/6
12	$>$1h/17	802.56/17	3.89/6	0.51/6
13	$>$1h/18	1793.08/18	6.16/6	0.73/6
14	$>$1h/18	$>$1h/18	6.44/6	0.70/6

The first column in each table give the number of philosophers and the number of the concurrent processes trying to access the critical section, respectively. The next four are:

- execution time of BCSat (in seconds) / the bound used with interleaving semantics
- execution time of ZChaff (in seconds) / the bound used with interleaving semantics
- execution time of BCSat (in seconds) / the bound used with step semantics
- execution time of ZChaff (in seconds) / the bound used with step semantics

In Table 3 the bound (number of steps) is the smallest number with which a deadlock was found. It turned out to be the number

[5] the translator `bc2cnf` available from `http://www.tcs.hut.fi/~tjunttil`

of philophers in the interleaving case whereas a single step sufficed in the step case. In Table 4 the mutex protocol satisfied the verified safety property and thus no model could be obtained. The bounds used are based on considerations of the diameter [4] of the associated Kripke structure. In the interleaving case the value of $\#proc + 5$ was used whereas using step semantics the constant value of 6 can be justified. All the models are available from `http://www.tcs.hut.fi/~tjussila/promelab`.

7 CONCLUSIONS

This paper presents ongoing work towards developing bounded model checking techniques for verifying asynchronous protocol and control software. A simple parallel programming language (a subset of PROMELA) is taken as the starting point and as the first step a compact propositional encoding of bounded executions of a parallel program is devised. The idea is to enhance the efficiency of bounded model checking by exploiting the potential concurrency available when executing an asynchronous program. As the first result it is shown how to capture both the standard interleaving executions and the step executions (allowing performing multiple statements simultaneously) using a propositional encoding.

Future work includes the study of other execution models allowing concurrency. One interesting alternative is the process semantics which has already been applied to reachability checking of 1-safe Petri nets [6]. In addition, the class of properties considered can be extended to the temporal logic LTL (without the next time operator, $\mathbf{X}$, in case of step semantics). Already Biere et. al [4] present an encoding of the temporal operators. However, the encoding is quadratic and may be improved, e.g., using ideas presented in [7].

REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers, Principles, Techniques and Tools*, Addison Wesley, 1986.

[2] M. Baldamus and J. Schröder-Babo, 'p2b: A translation utility for linking PROMELA and symbolic model checking (tool paper)', in *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, number 2057 in Lecture Notes in Computer Science, pp. 183–191. Springer Verlag, (2001).

[3] B. Bérard, Bidoit M., Finkel A, Laroussinie F., Petit A., Petrucci L., Ph. Schnoebelen, and McKenzie P., *Systems and Software Verification - Model-Checking Techniques and Tools*, Springer, 2001.

[4] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, 'Symbolic model checking without BDDs', in *Tools and Algorithms for the Construction and Analysis of Systems*, number 1579 in Lecture Notes in Computer Science, pp. 193–207, (1999).

[5] Per Bjesse, Tim Leonard, and Abdel Mokkedem, 'Finding bugs in Alpha microprocessor using satisfiability solvers', in *Proceedings of the 13th International Conference of Computer-Aided Verification*, number 2102 in Lecture Notes in Computer Science, pp. 454–464, (2001).

[6] Keijo Heljanko, 'Bounded reachability checking with process semantics', in *Proceedings of the 12th International Conference on Concurrency Theory*, pp. 218–232, (August 2001).

[7] Keijo Heljanko and Ilkka Niemelä, 'Bounded LTL model checking with stable models', in *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, (September 2001).

[8] Gerard J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1991.

[9] Tommi A. Junttila and Ilkka Niemelä, 'Towards an efficient tableau method for boolean circuit satisfiability testing', in *Computational logic - CL2000*, number 1861 in Lecture Notes in Artificial Intelligence, pp. 553–567, (2000).

[10] H. Kautz and B. Selman, 'Planning as satisfiability', in *Proceedings of the 10th Conference on Artificial Intelligence*, pp. 359–363, (1992).

[11] Fred Kröger, *Temporal Logic of Programs*, Springer Verlag, 1987.

[12] J. McCarthy and P.J. Hayes, 'Some philosophical problems from the standpoint of artificial intelligence', in *Machine Intelligence 4*, 463–502, Edinburgh University Press, Edinburgh, (1969).