Bounded Model Checking of Compositional Processes

Jun Sun, Yang Liu and Jin Song Dong
School of Computing,
National University of Singapore
{sunj,liuyang,dongjs}@comp.nus.edu.sg

Jing Sun
Department of Computer Science
The University of Auckland
j.sun@cs.auckland.ac.nz

Abstract

Verification techniques like SAT-based bounded model checking have been successfully applied to a variety of system models. Applying bounded model checking to compositional process algebras is, however, not a trivial task. One challenge is that the number of system states for process algebra models is not statically known, whereas exploring the full state space is computationally expensive. This paper presents a compositional encoding of hierarchical processes as SAT problems and then applies state-of-the-art SAT solvers for bounded model checking. The encoding avoids exploring the full state space for complex systems so as to deal with state space explosion. We developed an automated analyzer which combines complementing model checking techniques (i.e., bounded model checking and explicit on-the-fly model checking) to validate system models against event-based temporal properties. The experiment results show the analyzer handles large systems.

1. Introduction

Formal verification reveals inconsistencies of the specification and thus improves the reliability of the product. The notion of *model checking* [14] has been widely accepted as a successful means of formal verification [9, 5, 2]. The idea is to exhaustively explore all reachable states of a finite state machine (which represents an abstract view of a system) so as to tell whether a desired property is guaranteed or not. The original proposal of model checking relies on exhaustive search through explicit representations of reachable system states [14], which is known as *explicit model checking*. It suffers from the *state space explosion* problem. Later, *symbolic model checking* was proposed to overcome this problem by enumerating states symbolically (typically based on the notion of BDDs [10]). However, human intervention may be required to fine-tune the variable ordering so as to reduce the size of BDDs. In recent years, *bounded model checking* [13] have been proposed to complement explicit model checking and symbolic model checking with great success. The idea is to encode finite state machines (as well as the properties to be verified) as a Boolean formula that is satisfiable if and only if the underlying state machine can realize a finite sequence of transitions that reaches states of interest, and then apply state-of-the-art SAT solvers [1] to produce counterexamples (if any) efficiently. If such a path segment cannot be found at a given length *k*, the search is continued for larger *k*. With the rapid development of SAT solvers, we believe bounded model checking is promising for formal verification.

Previous works on model checking have been historically centered around state machines. Model checking techniques have only been applied to event-based formalisms to a limited extent. To our best knowledge, bounded model checking has not yet been applied to event-based languages like Communicating Sequential Processes (CSP [19]) or CCS. One of the reasons is that unlike in circuit verification [12] (where encoding the transition relation is rather straightforward), encoding the semantics of compositional processes using Boolean formulae is nontrivial. The number of system states for process algebra models is not statically known and exploring the full state space is computationally expensive. This paper presents a compositional encoding of hierarchical processes as SAT problems. State-of-the-art SAT solvers are then applied for bounded model checking. The encoding avoids exploring the full state space for complex systems so as to avoid state space explosion. Based on the idea, a toolkit has been developed to support formal system specification, simulation and verification (against temporal properties). The toolkit includes the two complementing model checkers, i.e., an explicit model checker and a bounded model checker. The advantages of applying bounded model checking instead of symbolic model checking include that SAT tools usually need far less hand manipulation than BDDs. The experiment results show that our toolkit has a competitive performance for verifying systems with large number of states.

The remainder of the paper is organized as follows. Section 2 briefly introduces the specification language we are dealing with. Section 3 presents how to encode semantics of compositional processes as Boolean formulae at the same time avoiding state space explosion. Section 4 introduces the functionalities of our tool in details. Section 5 concludes this paper.

2. Background

Without loss of generality, we present our idea in the setting of the classic CSP (in which multi-threaded alphabetized parallel plays an important role). A process is defined by the following syntax,

$$\begin{aligned} P \mathrel{\widehat{=}} & \; Stop_A \mid Skip \mid Run_A \mid e \rightarrow P \mid c?v \rightarrow P \\ \mid & \; c!x \rightarrow P \mid P;\; P \mid P \,\Box\, P \mid P \sqcap P \mid P \bigtriangleup P \\ \mid & \; P \lhd b \rhd Q \mid P \setminus A \mid P \parallel P \mid P|[A]|P \\ \mid & \; P \mid\mid\mid P \mid \|_i P_i \mid |||_i P_i \end{aligned}$$

where A is a set of events, e is an event, b is a Boolean expression and i is an index. Process $Stop_A$ never engages in any event from the set A. Process $Skip$ terminates successfully. Run_A may perform any sequence of event as long as the events are from A. Action prefixing $e \rightarrow P$ is initially willing to engage in event e and behaves as P afterward. Note that $Skip \mathrel{\widehat{=}} \checkmark \rightarrow Stop$ where $\widehat{=}$ means "is defined as" and $\checkmark$ is a special event denoting *termination*. The sequential composition, $P_1;\; P_2$, behaves as P_1 until its termination and then behaves as P_2. One way to introduce diversity of behaviors is through choices. A choice between two processes is denoted as $P_1 \,\Box\, P_2$ (for external choice) or $P_1 \sqcap P_2$ (for internal choice). $P_1 \bigtriangleup P_2$ behaves as P_1 until the first event of P_2 is engaged, then P_1 is interrupted and P_2 takes control. Process $P \lhd b \rhd Q$ behaves as P is b evaluates to true. Otherwise, it behaves as Q. Note that a process can be parameterized in the standard way. A parameter has the scope of the whole process expression. b is over variables in scope (as well as possible inputs). Process $P \setminus A$ hides observational of occurrences of events from A. Recursion is allowed by process referencing. The semantics of recursion follows Scott's fixed-point theory.

Let Σ be the set of all visible events, which excludes τ (invisible event) and $\checkmark$. Let $\alpha P \subseteq \Sigma$ be the alphabet of P. Parallel composition of two processes is written as $P_1 \parallel P_2$, where common events ($\alpha P_1 \cap \alpha P_2$) of P_1 and P_2 are synchronized. Interleaving is written as $P \mid\mid\mid Q$. The general form of parallel composition is $P\,|[\,A\,]|\,Q$ where events in A are synchronized by P and Q. Note that $P\,|[\,\alpha P \cap \alpha Q\,]| \equiv P \parallel Q$ and $P\,|[\,\varnothing\,]|\,Q \equiv P \mid\mid\mid Q$. The indexed version of interleaving and parallel composition is written as $|||_i P_i$ and $\|_i P_i$ respectively. The alphabet of a process can be separately defined or otherwise it is the set of events which constitute the process expression.

For simplicity and also the nature of model checking, we focus on the operational semantics in this paper. The sets of processes behaviors can equally and equivalently be extracted from the operational semantics, thanks to congruence theorems. The set of relevant transition rules can be found in [7] or [21]. Let $\overset{a}{\Rightarrow}$ be the transition relation defined by the operational semantics.

Definition A labeled transition system (LTS) is a 3-tuple (S, I, T) where S is a set of states, I is an initial state and $T : S \times \Sigma \times S$ is a labeled transition relation.

The language of an LTS is the set of (finite or infinite) runs which start with the initial condition and conform to the transition relation. The following defines the semantics of a process as an LTS. The language defined by the process is that of the LTS.

Definition Let P be a process. The semantics of P is defined as an LTS $\mathcal{L}^P = (S, I, T)$ where S is the set of all reachable processes, I is the initial process P and $T : S \times \alpha P \times S$ is the smallest transition relation such that $(P_1, a, P_2) \in T \Leftrightarrow P_1 \overset{a}{\Rightarrow} P_2$.

3. Encoding of Processes

This section is dedicated to a discussion on how to encode a given process P as Boolean formulae for bounded model checking. We start with encoding simple processes by explicitly building $\mathcal{L}^P$ and then discuss how to encode processes for which building $\mathcal{L}^P$ is not feasible. We remark that the encoding techniques is not restricted to CSP.

3.1. Encoding Simple Processes

A process P can be encoded by firstly constructing $\mathcal{L}^P$ and then encoding $\mathcal{L}^P$. Given an LTS, a property to verify and a bound k, we need to translate the LTS and the negation of the property into a propositional formula which is satisfiable if and only if there is a trace of length k which violates the property (i.e., a counterexample). Thus, we need to find an efficient encoding of states, events, and the transition relation. Given $\mathcal{L} = (S, I, T)$, we need $\lceil \log_2 \#S \rceil$ Boolean variables to encode the states. Let $\overrightarrow{xs}_i = \langle xs_i^1, xs_i^2, \cdots \rangle$ be a finite sequence of Boolean variables used to encode the states reached after $i-1$ steps. The encoding of a state is a Boolean formula π over $\overrightarrow{xs}_i$ such that $\pi(\overrightarrow{xs}_i) = 1$ if and only if the valuation of the variables uniquely identifies the state. Or equivalently, a state is associated with a unique binary number and each Boolean variable represents one bit of the number. Similarly, we use $\lceil \log_2 \#\alpha\mathcal{L} \rceil$ Boolean variables to encode the alphabet of $\mathcal{L}$. Let $\overrightarrow{xe}_i$ be the variables used to encode the events. A transition is encoded as a Boolean formula of the following form,

$$\pi(\overrightarrow{xs}_i) \wedge \pi(\overrightarrow{xe}_i) \wedge \pi(\overrightarrow{xs}_{i+1})$$

where $\overrightarrow{xs}_{i+1}$ is a set of fresh variables used to encode the post-state. Let Π denote the encoding function which maps a state or event to a Boolean formula given a set of Boolean variables. $\Pi(P1, \overrightarrow{xs}_i) \wedge \Pi(e, \overrightarrow{xe}_i) \wedge \Pi(P_2, \overrightarrow{xs}_{i+1})$ where $(P_1, e, P_2) \in T$ is the Boolean coding of the transition in the above form. Informally, this formula guarantees that if the transition is to be taken, the pre/post-state and event must be P_1/P_2 and e respectively. The transition relation T is encoded as the disjunction of all possible transitions, i.e., any encoded transition may be taken if it can be satisfied.

$$\mathcal{T}_i = \bigvee\{\Pi(P1, \overrightarrow{xs}_i) \wedge \Pi(e, \overrightarrow{xe}_i) \wedge \Pi(P_2, \overrightarrow{xs}_{i+1}) \mid (P_1, e, P_2) \in T\}$$

Given a bound k for bounded model checking, the encoded transition relation must be applied k-times. Every time a fresh set of variables must be used to encode the engaged event as well as the target state. Thus, we need $(k+1) \times \lceil \log_2 \#S \rceil + k \times \lceil \log_2 \#\Sigma \rceil$ Boolean variables to represent state $s_1 \mathinner{..} s_{k+1}$ and $e_1 \mathinner{..} e_k$ where $s_1 = I$.

Definition An encoding of an LTS is 4-tuple $\mathcal{E} = (\mathcal{I}, \mathcal{T}_i, \overrightarrow{xs}_i, \overrightarrow{xe}_i)$ where $\mathcal{I} = \Pi(I, \overrightarrow{xs}_1)$ is the encoded initial state, $\mathcal{T}_i$ is the encoded transition relation as defined above, $\overrightarrow{xs}_i$ are the variables used to encode the source state of $\mathcal{T}_i$ and $\overrightarrow{xe}_i$ are the variables used to encode the labeling events of transitions of $\mathcal{T}_i$.

Given an LTS $\mathcal{L}$ and its encoding $\mathcal{E}$, we say $\mathcal{E}$ is sound if and only if $\mathcal{E}$ and $\mathcal{L}$ are trace-equivalent, i.e., every trace allowed by $\mathcal{L}$ must be allowed by $\mathcal{E}$ and *vice versa*. The above encoding of an LTS is sound as we can show that the encoded transition relation conforms to T and the encoded initial condition conforms to I. Given an encoding of the system $\mathcal{E}$, a property ϕ to verify and a bound k, the propositional formula constructed is of the following form,

$$[\![\mathcal{E}, \phi]\!]_k \mathrel{\widehat=} \mathcal{I} \wedge \bigwedge_{i=1}^{k} \mathcal{T}_i \wedge [\![\neg\, \phi]\!]_k$$

where $[\![\neg\, \phi]\!]_k$ is the encoded negation of the given property (with regards to k). We leave it to Section 4 for detailed discussion. A satisfiability solution to the above formula gives a counterexample of the property, which satisfies the initial condition and the transition relation up to k-steps and violates the property.

In the following, we write $\mathcal{E}^P$ to denote the encoding of P. Explicitly constructing $\mathcal{L}^P = (S^P, I^P, T^P)$ is however not always desirable for several reasons. Firstly, S^P (and therefore T^P) may not be finite. For instance, processes like $P \mathrel{\widehat=} b \rightarrow Skip \,\Box\, (a \rightarrow P;\ c \rightarrow Skip)$ or $P \mathrel{\widehat=} a \rightarrow (P \mid\mid\mid P)$ allow unbounded recursion or replication and, thus, may result in infinite reachable process expressions. Our experiences, however, show that processes of the above forms are rather rare in practice. Without loss of generality, we assume that S^P is always finite. Optimally, the number of Boolean variables needed to encode S^P is $\lceil \log_2 \#S^P \rceil$. However, determining the exact size of S^P requires traversing through all reachable states, which is often undesirable due to state space explosion. For instance, assume $\#S^Q = n$, the interleaving of m copies of Q (say P) has n^m states. One remedy is to encode the $\mathcal{L}^Q$ (if its size is manageable) and then compose $\mathcal{E}^Q$ to generate $\mathcal{E}^P$ so as to avoid constructing $\mathcal{L}^P$.

3.2. Composing Encodings

A rich set of operators can be used to compose processes as illustrated in Section 2. Among all operators, it is the indexed parallel composition or indexed interleaving (which we refer to as indexed concurrency) which causes state space explosion. Given P which contains indexed concurrency, instead of building $\mathcal{L}^P$ we shall deduce $\mathcal{E}^P$ from the encoding of its sub-components. In the following, we show how to compose the encoding of sub-components for various composition. In order to draw connections between transitions of different processes running in parallel, a global event-to-Boolean encoding is established beforehand. In the following, let $\Pi(e, \overrightarrow{xe})$ be the formula encoding e using variables $\overrightarrow{xe}$. Given $\overrightarrow{xs}_i = \langle xs_i^1, xs_i^2, \cdots, xs_i^n \rangle$ where $i \in \{1, 2\}$ as two sequences of Boolean variables of the same length, we write $\overrightarrow{xs}_1 \Leftrightarrow \overrightarrow{xs}_2$ to mean $xs_1^1 \Leftrightarrow xs_2^1 \wedge xs_1^2 \Leftrightarrow xs_2^2 \wedge \cdots \wedge xs_1^n \Leftrightarrow xs_2^n$. To further abuse notations, we write $\overrightarrow{xs}_1 \cup \overrightarrow{xs}_2$ to denote the sequence of variables which contains both variables in $\overrightarrow{xs}_1$ and $\overrightarrow{xs}_2$ and is then sorted (according to the unique variables ID).

Definition Let $P = \mid\mid\mid_{j=1}^{n} P_j$. Let $\mathcal{E}^{P_j} = (\mathcal{I}^{P_j}, \mathcal{T}_i^{P_j}, \overrightarrow{xs}_i^{P_j}, \overrightarrow{xe}_i^{P_j})$. The encoding of P is $(\mathcal{I}^P, \mathcal{T}_i^P, \overrightarrow{xs}_i^P, \overrightarrow{xe}_i^P)$ where $\mathcal{I}^P = \bigwedge_{j=1}^{n} \mathcal{I}^{P_j}$, $\overrightarrow{xs}_i^P = \bigcup_{j=1}^{n} \overrightarrow{xs}_i^{P_j}$, $\overrightarrow{xe}_i^P = \overrightarrow{xe}_i^{P_j}$ and $\mathcal{T}_i^P = \bigvee_{j=1}^{n} (\mathcal{T}_i^{P_j} \wedge \bigwedge_{m \neq j} (\overrightarrow{xs}_i^{P_m} \Leftrightarrow \overrightarrow{xs}_{i+1}^{P_m}))$.

Note that the variables used to encode each P_j are disjoint. The encoded initial condition of P is the conjunction of the encoded initial conditions of each sub-component. Intuitively, this says that when the composition is initialized, all sub-components must be at its initial state. The predicate $\overrightarrow{xs}_i^{P_m} \Leftrightarrow \overrightarrow{xs}_{i+1}^{P_m}$ means that P_m remains in the same state. The encoded transition relation is the disjunction of a set of clauses, each of which states that a transition of P_j may be taken and the states of other sub-components are unchanged. Thus, any transition of a sub-component can be taken without affecting other sub-components. Indexed parallel composition is handled similarly. The complication is that the alphabet of a sub-component may actually contain more events than those constitute the process expression. The following definition shows that by manipulating the encoded transition relations of the sub-components, the encoded transition relation of the composition shall be exactly the conjunction of the encoded transition relations of the sub-components.

Definition Let $P = \|_{j=1}^{n} P_i$. Let $\mathcal{E}^{P_j} = (\mathcal{I}^{P_j}, \mathcal{T}_i^{P_j}, \overrightarrow{xs}_i^{P_j}, \overrightarrow{xe}_i^{P_j})$. The encoding of P is $(\mathcal{I}^P, \mathcal{T}_i^P, \overrightarrow{xs}_i^P, \overrightarrow{xe}_i^P)$ where $\mathcal{I}^P = \bigwedge_{j=1}^{n} \mathcal{I}^{P_j}$, $\overrightarrow{xs}_i^P = \bigcup_{j=1}^{n} \overrightarrow{xs}_i^{P_j}$, $\overrightarrow{xe}_i^P = \overrightarrow{xe}_i^{P_j}$, and $\mathcal{T}_i^P$ is defined as follows,

$$\begin{aligned}\mathcal{T}_i^P = \textstyle\bigwedge_{j=1}^{n}(\mathcal{T}_i^{P_j} \vee \bigvee\{\Pi(e, \overrightarrow{xe}_i) \wedge \overrightarrow{xs}_i \Leftrightarrow \overrightarrow{xs}_{i+1} \mid \\ e \notin \alpha P_j \wedge e \in \alpha P \cup \{\tau\}\})\end{aligned}$$

The transition relation $\mathcal{T}_i^{P_j}$ is extended with clauses to allow events not in αP_i but in αP (including τ) to occur freely without changing the status of this sub-component. Because the encoded transition relation of each sub-component is conjuncted and we use the same set of variables to encode the events, an event can be engaged if and only if every sub-component participates in it. This construction guarantees that the encoded transition relation of the composition allows only runs which conform to the semantics. It avoids constructing $\mathcal{L}^P$ by paying the price of extra transitions.

Example The following specifies the classic dining philosophers problem [19],

$$\begin{aligned}
Phil(i) &= think.i \rightarrow get.i.(i+1)\%N \rightarrow get.i.i \\
&\quad \rightarrow eat.i \rightarrow put.i.(i+1)\%N \rightarrow put.i.i \\
&\quad \rightarrow Phil(i) \\
Fork(i) &= get.i.i \rightarrow put.i.i \rightarrow Fork(i) \ \Box \\
&\quad get.((i-1)\%N).i \rightarrow put.((i-1)\%N).i \\
&\quad \rightarrow Fork(i) \\
Phils(N) &= \|_{i=0}^{N-1} (Phil(i) \parallel Fork(i))
\end{aligned}$$

where N is the number of philosophers, $get.i.j$ ($put.i.j$) is the action of the i-th philosopher picking up (putting down) the j-th fork. Assuming $N = 5$ and x_1, x_2, x_3 are used to encode the events, the event encoding is shown in the following table (and the rest are ignored for brevity).

Event	Encoding	Event	Encoding
$think.0$	$\neg x_1 \wedge \neg x_2 \wedge \neg x_3$	$put.0.1$	$x_1 \wedge \neg x_2 \wedge \neg x_3$
$get.0.1$	$\neg x_1 \wedge \neg x_2 \wedge x_3$	$put.0.0$	$x_1 \wedge \neg x_2 \wedge x_3$
$get.0.0$	$\neg x_1 \wedge x_2 \wedge \neg x_3$	$get.4.0$	$x_1 \wedge x_2 \wedge \neg x_3$
$eat.0$	$\neg x_1 \wedge x_2 \wedge x_3$	$put.4.0$	$x_1 \wedge x_2 \wedge x_3$

The following is the encoded transition relation $\mathcal{T}_1^{Phil(0) \| Fork(0)}$.

$$\begin{aligned}
&\mathcal{T}_1^{Phil(0)} \wedge \mathcal{T}_1^{Fork(0)} \\
&\vee (x_1 \wedge x_2 \wedge \neg x_3 \wedge x_4 \Leftrightarrow x_7 \wedge x_5 \Leftrightarrow x_8 \wedge x_6 \Leftrightarrow x_9) \\
&\vee (x_1 \wedge x_2 \wedge x_3 \wedge x_4 \Leftrightarrow x_7 \wedge x_5 \Leftrightarrow x_8 \wedge x_6 \Leftrightarrow x_9)) \\
&\vee (\neg x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge x_{10} \Leftrightarrow x_{12} \wedge x_{11} \Leftrightarrow x_{13}) \\
&\vee (\neg x_1 \wedge \neg x_2 \wedge x_3 \wedge x_{10} \Leftrightarrow x_{12} \wedge x_{11} \Leftrightarrow x_{13})) \\
&\vee (\neg x_1 \wedge x_2 \wedge x_3 \wedge x_{10} \Leftrightarrow x_{12} \wedge x_{11} \Leftrightarrow x_{13})) \\
&\vee (x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge x_{10} \Leftrightarrow x_{12} \wedge x_{11} \Leftrightarrow x_{13}))
\end{aligned}$$

where x_4, x_5, x_6 (x_7, x_8, x_9) are used to encode the pre-state (post-state) of $Phil(0)$ and x_{10}, x_{11} (x_{12}, x_{13}) are used to encode the pre-state (post-state) of $Fork(0)$. □

A large class of systems can be specified as an indexed parallel composition or indexed interleaving of multiple sub-components which have relatively small number of states, e.g., $Phils_N$ is specified as an indexed parallel composition of philosopher and fork fairs. For such systems, we encode each sub-component by explicitly constructing the LTS and then apply the above construction to build the composed transition relation. Nonetheless, if a process P which contains indexed concurrency is further composed with other processes using operators like $\triangle$, $\Box$ and ; , or amended using operators like $\backslash$, we shall be able to deduce the encoding of the composition from the encoding of P (and others). For instance, assuming the given process is $Phils(N)$; Q, we must not explore all states of $Phils(N)$ in order to encode the given process.

Definition Let $P = M \ \Box \ N$. Let $\mathcal{E}^M = (\mathcal{I}^M, \mathcal{T}_i^M, \overrightarrow{xs}_i^M, \overrightarrow{xe}_i^M)$ be the encoding of M. Let $\mathcal{E}^N = (\mathcal{I}^N, \mathcal{T}_i^N, \overrightarrow{xs}_i^N, \overrightarrow{xe}_i^N)$. The encoding of P is $(\mathcal{I}^P, \mathcal{T}_i^P, \overrightarrow{xs}_i^P, \overrightarrow{xe}_i^P)$ where $\mathcal{I}^P = \mathcal{I}^M \wedge \mathcal{I}^N$, $\overrightarrow{xs}_i^P = \overrightarrow{xs}_i^M \cup \overrightarrow{xs}_i^N \cup \{xc_i\}$, $\overrightarrow{xe}_i^P = \overrightarrow{xe}_i^M = \overrightarrow{xe}_i^N$, and $\mathcal{T}_i^P = (xc_i \wedge \mathcal{T}_i^M \wedge xc_{i+1}) \vee (\neg xc_i \wedge \mathcal{T}_i^N \wedge \neg xc_{i+1})$ where xc_i is a fresh control variable.

The encoded initial condition has no constraints on xc_1 and thus xc_1 can be either true or false initially (which means transitions from P or Q can be taken). Once one of the choices has been taken, xc_i remains the same as xc_1 for all i and thus a later step must respect the choice made at the first step. This captures the semantics of choices. Note that $\sqcap$ is handled in the same way as $\Box$ and $\sqcap$ are equivalent in the trace semantics [19].

Definition Let $P = M \ \triangle \ N$. Let $\mathcal{E}^M = (\mathcal{I}^M, \mathcal{T}_i^M, \overrightarrow{xs}_i^M, \overrightarrow{xe}_i^M)$ be the encoding of M. Let $\mathcal{E}^N = (\mathcal{I}^N, \mathcal{T}_i^N, \overrightarrow{xs}_i^N, \overrightarrow{xe}_i^N)$. The encoding of P is $(\mathcal{I}^P, \mathcal{T}_i^P, \overrightarrow{xs}_i^P, \overrightarrow{xe}_i^P)$ where $\mathcal{I}^P = \mathcal{I}^M \wedge \mathcal{I}^N$, $\overrightarrow{xs}_i^P = \overrightarrow{xs}_i^M \cup \overrightarrow{xs}_i^N \cup \{xc_i\}$, $\overrightarrow{xe}_i^P = \overrightarrow{xe}_i^M = \overrightarrow{xe}_i^N$, and $\mathcal{T}_i^P = (\neg xc_i \wedge \mathcal{T}_i^M \wedge \neg xc_{i+1}) \vee (\mathcal{T}_i^N \wedge xc_{i+1})$ where xc_i is a fresh control variable.

Interrupt can be viewed as a biased choice. Note that $\neg xc_i$ is true if and only if M has not yet been interrupted. If a transition of T_M is taken, xc_{i+1} remains false so that next transition can be taken from T_M or T_N. Whenever a transition of T_N is taken, xc_{i+1} must be true, which forbids all transitions from T_M.

Definition Let $P = M;\ N$. Let $\mathcal{E}^M = (\mathcal{I}^M, \mathcal{T}_i^M, \overrightarrow{xs}_i^M, \overrightarrow{xe}_i^M)$ be the encoding of M. Let $\mathcal{E}^N = (\mathcal{I}^N, \mathcal{T}_i^N, \overrightarrow{xs}_i^N, \overrightarrow{xe}_i^N)$. The encoding of P is $(\mathcal{I}^P, \mathcal{T}_i^P, \overrightarrow{xs}_i^P, \overrightarrow{xe}_i^P)$ where $\mathcal{I}^P = \mathcal{I}^M \wedge \neg xc_1$, $\overrightarrow{xs}_i^P = \overrightarrow{xs}_i^M \cup \overrightarrow{xs}_i^N \cup \{xc_i\}$, $\overrightarrow{xe}_i^P = \overrightarrow{xe}_i^M = \overrightarrow{xe}_i^N$, and $\mathcal{T}_i^P$ is defined as follows: where xc_i is a fresh variable,

$$\begin{aligned}\neg xc_i \wedge \mathcal{T}_i^M \wedge (\neg \Pi(\checkmark, \overrightarrow{xe}_i) \wedge \neg xc_{i+1} \vee \\ \Pi(\checkmark, \overrightarrow{xe}_i) \wedge xc_{i+1} \wedge \mathcal{I}^N) \vee xc_i \wedge \mathcal{T}_i^N\end{aligned}$$

Initially, $\neg xc_1$ must be true. Note that $\neg xc_i$ is true if and only if M has not yet terminated. Intuitively, a sequential composition can be viewed a delayed choice whereby transitions from N can only be taken after a $\checkmark$ transition has been taken in M. xc_i and $\mathcal{I}^N$ is true once a transition of M labeled with $\checkmark$ has been taken. Because transitions from M are guarded with $\neg xc_i$, no transition from M can be taken afterwards.

Other compositional operators are handled similarly by manipulating the encoded transition relations of the sub-components and introducing control variables if necessary. For instance, if some events of an encoded process are to be hidden (i.e., $P \backslash A$), those events are renamed, i.e., the label e of a transition is encoded as $\Pi(\tau_e, \overrightarrow{xe_i})$ instead of $\Pi(e, \overrightarrow{xe_i})$. Note that hiding different events results in different τ transitions. This prevents synchronization between different hidden events.

Given a process P, if P contains no indexed concurrency, we construct $\mathcal{L}^P$ and then apply the encoding in Section 3.1. Otherwise, each sub-component of the indexed interleaving or parallel composition is encoded first. The encoding of P is then composed by applying the compositional encoding. If a sub-component of the indexed interleaving or parallel (say Q) contains indexed concurrency as well, the same procedure is repeated so as to encode Q. Note that for processes like $P = a \rightarrow P \mid\mid\mid \cdots \mid\mid\mid P$, this construction is not feasible (and thus we have to construct $\mathcal{L}^P$). Nonetheless, for most interesting systems in which there is no unbounded replication (or recursion), this construction not only terminates but results in Boolean formulae of manageable size, which can be efficiently solved by SAT-solvers.

Theorem 3.1 *Let P be a process. $\mathcal{E}^P$ is the encoding of P as defined above. $\mathcal{E}^P$ is trace-equivalent to $\mathcal{L}^P$.*

This theorem states that our encoding is sound. It is proved by structural induction. The base case is when a process contains no indexed concurrency, i.e., the encoding in Definition 3.1 is sound. Then we prove the induction step by showing the compositional encoding preserves the equivalence. We skipped the proof for brevity.

4. The Process Analysis Toolkit

We developed an analyzer to apply bounded model checking and on-the-fly explicit model checking. The analyzer has been implemented in C# and is publicly available at our web site [23]. It consists of three main components: a specification editor, a simulator and two complementing model checkers. The editor provides a user friendly interface (with featured text editing, syntax highlighting, multi threading execution, multi-documents environment, etc) for users to introduce system models (i.e., the full CSP syntax is supported) as well as desirable properties (i.e., the full temporal logic syntax is supported). The system models are parsed into internal representations, which are used for both simulation and verification. The simulator takes in the system models and allows users to perform various simulation tasks: complete states generation based on the execution graph, random simulation, user interactive simulation, trace replay and so on. In this section, we focus on the event-based on-the-fly/bounded model checking.

4.1. Temporal Properties

We support temporal logic verification, which complements the established tools like FDR [21] (which verifies process refinement relationships). Because we are dealing with an event-based formalism, we extend standard Linear Temporal Logic (LTL) with events so that properties concerning both states and events can be stated and verified. The extended LTL is as follows.

Definition Let Pr be a set of propositions. An extended LTL formula is,

$$\phi ::= p \mid a \mid \neg\phi \mid \phi \wedge \psi \mid \bigcirc\phi \mid \Box\phi \mid \Diamond\phi \mid \phi U \psi$$

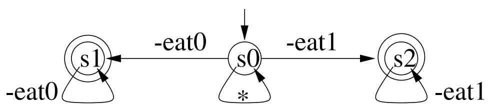
where p ranges over Pr and a ranges over Σ. Let $\pi = \langle P_0, x_0, P_1, x_1, \cdots \rangle$ be an infinite sequence of events. Let π^i be the suffix of π starting in P_i.

$$\begin{array}{ll}
\pi^i \models p & \Leftrightarrow P_i \models p \\
\pi^i \models a & \Leftrightarrow x_{i-1} = a \\
\pi^i \models \neg\phi & \Leftrightarrow \neg(\pi^i \models \phi) \\
\pi^i \models \phi \wedge \psi & \Leftrightarrow \pi^i \models \phi \wedge \pi^i \models \psi \\
\pi^i \models \bigcirc\phi & \Leftrightarrow \pi^{i+1} \models \phi \\
\pi^i \models \Box\phi & \Leftrightarrow \forall j \geq i \bullet \pi^j \models \phi \\
\pi^i \models \Diamond\phi & \Leftrightarrow \exists j \geq i \bullet \pi^j \models \phi \\
\pi^i \models \phi U \psi & \Leftrightarrow \exists j \geq i \bullet \pi^j \models \psi \wedge \forall k \mid \\
 & \quad i \leq k \leq j-1 \bullet \pi^j \models \phi
\end{array}$$

Example The following specifies a desirable property of $Phils_N$: $\Box\Diamond eat_0 \wedge \Box\Diamond eat_1 \cdots \Box\Diamond eat_{N-1}$ where $\Box$ reads as "always" and $\Diamond$ reads as "eventually". The property states that every philosopher will always eventually eat, i.e., no one starves. □

The simplicity of writing formulas concerning events as in the above example is not purely a matter of aesthetics. It may yield gains in time and space [11]. Given an extended LTL formula, a trace equivalent Büchi automaton is constructed efficiently using the state-of-the-art conversion proposed in [18]. In other words, we adapt an automata-based approach for explicit LTL model checking as Spin [20]. Note that for efficient reasons, the Büchi automata are transition-labeled (instead of state-labeled).

Example The following is the Büchi Automaton generated from the negation of the formula $\Box\Diamond eat_0 \wedge \Box\Diamond eat_1$,



where $s0$ is the initial state, $s1$ and $s2$ are two accepting states and $*$ means the transition is unguarded. $\neg\, e$ means the transition can be labeled with any event but e. □

Let $B^{\neg\,\phi}$ be the Büchi automaton constructed from property $\neg\,\phi$. In the explicit model checking approach, the product of $B^{\neg\,\phi}$ and P is generated (same as in Spin). Explicit model checking is to determine the emptiness of $\mathcal{L}^P \times B$, i.e., explore on-the-fly whether the product contains a loop which is composed of at least one accepting state. Finite traces are extended to infinite ones in a standard way. In the presence of a counterexample, on-the-fly model checking usually produces a trace leading to a bad state or a loop quickly (refer to Section 4.2). However, the counterexample produced may be extremely long because it relies on a depth first search. Bounded model checking may then be used to produce a shorter trace which leads to the same bad state or loop. Though because of the Büchi automata, the generated counterexample may not be the shortest. Nonetheless, our bounded model checker can be used as a separate model checker.

For bounded model checking, because $\mathcal{L}^P$ may not be built, we encode the Büchi automaton and then compose it with the encoding of the model (i.e., $\mathcal{E}^P$). Given a guard g labeled with a transition of B, let $\Pi(g, \overrightarrow{xe}_i^B, \overrightarrow{xs}_{i+1}^B)$ be the encoded guard, e.g., $\Pi(e, \overrightarrow{xe}_i^B)$ if g is an event e or $\neg\,\Pi(e, \overrightarrow{xe}_i^B)$ if g is an event $\neg\, e$ or $\Pi(g_1, \overrightarrow{xe}_i^B, \overrightarrow{xs}_{i+1}^B) \wedge \Pi(g_2, \overrightarrow{xe}_i^B, \overrightarrow{xs}_{i+1}^B)$ if g is $g_1 \wedge g_2$.

Definition Let P be a process. Let $\mathcal{E}^P = (\mathcal{I}^P, \mathcal{T}_i^P, \overrightarrow{xs}_i^P, \overrightarrow{xe}_i)$. Let B be a Büchi automaton (S, I, T, F) . $\mathcal{E}^B = (\mathcal{I}^B, \mathcal{T}_i^B, \overrightarrow{xs}_i^B, \overrightarrow{xe}_i)$ where

$$\begin{aligned}\mathcal{T}_i^B = \bigvee\{&\Pi^B(s, \overrightarrow{xs}_i^B) \wedge \Pi(g, \overrightarrow{xe}_i^B, \overrightarrow{xs}_{i+1}^B) \wedge \\ &\Pi^B(s', \overrightarrow{xs}_{i+1}^B) \mid (s, g, s') \in T\}\end{aligned}$$

The encoding of the product of P and B is $(\mathcal{I}, \mathcal{T}_i, \overrightarrow{xs}_i, \overrightarrow{xe}_i)$ where $\mathcal{I} = \mathcal{I}^B \wedge \mathcal{I}^P$, $\overrightarrow{xs}_i = \overrightarrow{xs}_i^P \cup \overrightarrow{xs}_i^B$ and $\mathcal{T}_i = \mathcal{T}_i^P \wedge \mathcal{T}_i^B$.

Because P and B share the same alphabet as well as the variables to encode the events, transitions of P and B are always synchronized. P violates the property if and only if the language of $P \times B$ is not empty. Let $\mathcal{F}^B(\overrightarrow{xs}_i) = \bigvee\{\Pi^B(s, \overrightarrow{xs}_i) \mid s \text{ is an accepting state of B}\}$ be the encoded accepting states. The following theorem states the correctness of our bounded model checking.

Theorem 4.1 *Given a process P, and a Büchi automaton B constructed from $\neg\,\phi$, let $\mathcal{E}^{P\times B} = (\mathcal{I}, \mathcal{T}_i, \overrightarrow{xs}_i, \overrightarrow{xe}_i)$ be the encoding of $P \times B$. Let k to be a bound. The following formula is satisfiable iff there is a counterexample of size k.*

$$[\![P, \phi]\!]_k = \mathcal{I} \wedge \bigwedge_{i=1}^{k} \mathcal{T}_i \wedge [\![\neg\,\phi]\!]_k$$

where $[\![\neg\,\phi]\!]_k$ is $\bigvee_{i=1}^{k-1}\{\overrightarrow{xs}_k \Leftrightarrow \overrightarrow{xs}_i \wedge \bigvee_{j=i}^{k}\{\mathcal{F}^B(\overrightarrow{xs}_j)\}\}$.

The proof is sketched in the following. A solution to $[\![P, \phi]\!]_k$ is an assignment of *true* or *false* to $\bigcup_{i=1}^{k+1} \overrightarrow{xs}_i$ and $\bigcup_{i=1}^{k} \overrightarrow{xe}_i$ as well as the control variables (if any), from which we can identify a finite run $\langle s_1, e_1, s_2, e_2, \cdots, s_k, e_k, s_{k+1}\rangle$. Because $\mathcal{I}$ must be true, s_1 is an initial state. Because $\mathcal{T}_i$ must be true, by Theorem 3.1, $s_i \overset{e_i}{\Rightarrow} s_{i+1}$ for all $1 \leq i \leq k$. Thus, the sequence of states/events identified must be a run of P (as well as a finite prefix of a trace of ϕ). The constraint $[\![\neg\,\phi]\!]_k$ states that the finite run must contain a loop, i.e., $xs_k \Leftrightarrow xs_i$ for some i, and the loop must contain at least one accepting state, i.e., there exists some j satisfying $j \geq i \wedge j \leq k$ such that s_j is accepting. Therefore, the finite run identifies an infinite trace which is allowed by P and violates ϕ.

4.2. Performance Evaluation

In this section, we present a number of experiments to show the feasibility of applying SAT-based model checking to process algebras. The effectiveness of our compositional encoding is straightforward. If compositional encoding is applied, the encoding time is often negligible and the size of the formula is comparable to that of the formula generated by constructing the LTS[1].

For timely efficiency, we compare our analyzer with FDR and Spin. FDR is *de facto* model checker for CSP, which has been actively developed for years. We choose Spin over others because it is the most established explicit model checker and its input language is loosely based on CSP. Note that partial order reduction, which partly makes Spin very successful, has been implemented in our analyzer. We choose not to compare our bounded model checker with nuSMV because SMV focuses a different application domain (i.e., circuit verification), in which often the transition relation is known statically. Our bounded model checker has been evaluated with two award wining SAT solvers, i.e., MiniSAT and RSAT [1].

Figure 1 summarizes the performance using three benchmark models, i.e., the dining philosopher problem as in Example 3.2 (against the property in Example 4.1), the classic readers/writers problem and Milner's cyclic scheduler. This model describes a protocol for coordination of N readers and N writers accessing a shared resource. The property to verify is reachability of an erroneous situation (i.e., wrong readers/writers coordination). Milner's cyclic scheduler describes a scheduler for N concurrent processes. The processes are scheduled in cyclic fashion so that the first process is reactivated after the N-th process has been activated. The property to verify is that a process must eventually be scheduled. Details of the models and more experiments can be found at [23]. Note that the experiment re-

1 Depends on whether the processes are strongly coupled or not.

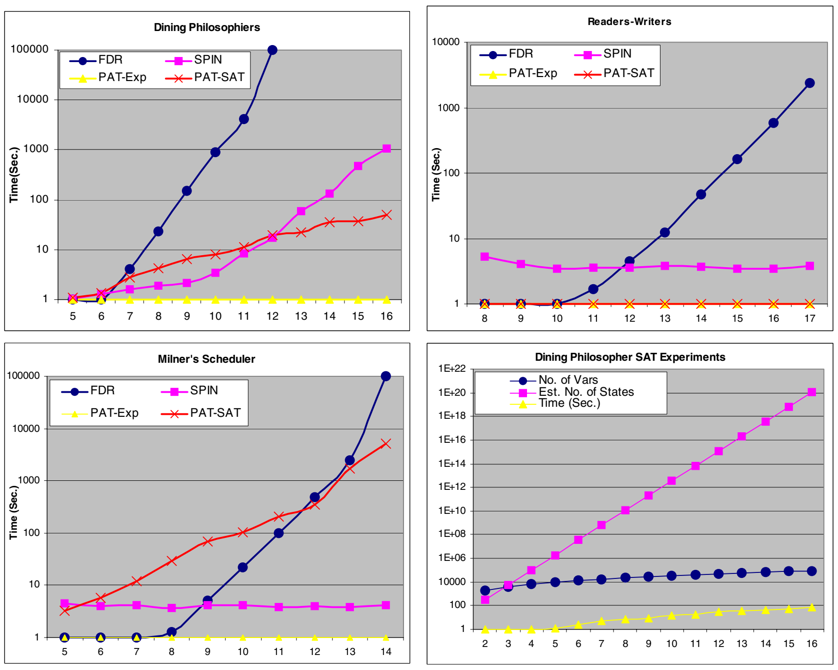


Figure 1. Performance Evaluation with a 2.0 GHz Intel Core Duo CPU and 1 GB memory

sults on FDR should be taken with a grain of salt, since the results are obtained by showing a (failure) refinement relationship between the system model and a process capturing the property to verify. For the dining philosopher example (the left-upper chart), our on-the-fly explicit model checker (referred as PAT-Exp) performs best to produce a counterexample. Our bounded model checker (referred as PAT-SAT) outperforms Spin for 13 or more philosophers. The main reason is that the LTL to Büchi automata conversion in Spin suffers from large LTL formulae, i.e., takes more time and produces bigger automata. All verifiers outperforms FDR (except PAT-SAT for small number of philosophers because of the encoding overhead), which is not feasible for more than 12 philosophers. For the readers/writers example, all verifiers except FDR produces a counterexample efficiently. Note that for every experiment Spin takes less than a few seconds to build model-specific executables. For the Milner's example, the full state space (which is exponentially increasing without partial order reduction) must be explored because the property to verify is true. SAT outperforms FDR for 12 or more processes. This suggests that SAT-based model checking has the potential to handle large state space. Moreover, the current implementation of PAT-SAT may be improved by orders of magnitude should we incorporate recently development on incremental bounded model checking and more [22, 24]. Nonetheless, bounded model checking currently is mainly for falsification (if without a proper threshold bound). The time taken by Spin and PAT-Exp remains constant. This should be credited to the partial order reduction. In the future, sophisticated optimization techniques like symmetry reduction will be incorporated into our analyzer. The right-bottom chart summarizes the performance our SAT-based verifier in terms of the size of the generated formula, the time needed for encoding and solving against the number of states of the model. The estimated number of states increase exponentially whereas the number of Boolean variables and the time needed increase much slower.

5. Conclusion and Future Works

In summary, we have developed a way to encode compositional system models without explicitly exploring all research states. In addition, a self-contained toolkit has been developed for system specification and verification. We have implemented a user-friendly environment for writing CSP specification, a simulator for examining and visualizing possible dynamic system behaviors, an explicitly model checker and a bounded model checker. Experiment results show that our analyzer does verification rather efficiently. Though presented in the framework of CSP, our encoding of compositional processes may be applied to other formal specification languages and notations. The toolkit is a starting point for verification support for formal specification languages and notations. We are extending the input language with more expressiveness power so as to broaden the application domain, i.e., arrays, global variables, etc. Since our toolkit is designed to be extensible, we plan to support more specification languages like CCS, π-calculus, Timed Automata (which requires SMT capability as in demonstrated in [3]) or integrated formalisms like Circus or TCOZ.

As for related works, our work is related to the line of verifiers developed in the formal methods community, e.g., FDR for CSP, Spin and nuSMV. There are relatively few analysis tools for CSP, despite its popularity and influence over quite a number of design languages such as Ada and *occam*. The noticeable ones include the model checker FDR [21], the simulator ProBE and those translators which transforms CSP models (or its extensions like Timed CSP) to other models so as to reuse existing verification mechanisms [6, 15, 16, 17]. FDR is the only model checker dedicated to CSP which we are aware of. FDR supports verification of process refinement relationship as well as deadlock/livelock-freeness. Compared to FDR, our toolkit allows system verification based on the full LTL, as well as deadlock-freeness checking, feasibility test, etc. In the future, we plan to extend our toolkit with the full functionalities of FDR. Spin is the most established explicit model checker, which has been applied widely. In literature, there have been a number of bounded model checkers dedicated to different specification languages. Some noticeable ones include the first bounded model checker [4], NuSMV [12] and UCLID [8]. However, as far as the authors know, there has not yet been a bounded model checker dedicated to process algebras, which have a compositional nature.

References

[1] SAT Competition. http://www.satcompetition.org/.

[2] R. Alur, L. J. Jagadeesan, J. J. Kott, and J. V. Olnhausen. Model-Checking of Real-Time Systems: A Telecommunications Application (Experience Report). In *Proc. of the 19th Inter. Conf. on Soft. Eng. (ICSE'97)*, pages 514–524, 1997.

[3] A. Armando, J. Mantovani, and L. Platania. Bounded Model Checking of Software Using SMT Solvers Instead of SAT Solvers. In *Proc. of the 13th Inte. SPIN Workshop on Model Checking Software (SPIN 2006)*, pages 146–162, 2006.

[4] A. Biere, A. Cimatti, E. M. Clarke, and Y. S. Zhu. Symbolic Model Checking without BDDs. In *Proc. of the 5th Inter. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, pages 193–207. Springer, 1999.

[5] A. Biere, E. M. Clarke, R. Raimi, and Y. S. Zhu. Verifiying Safety Properties of a Power PC Microprocessor Using Symbolic Model Checking without BDDs. In *Proc. of the 11th Inter. Conf. on Computer Aided Verification (CAV'99)*, pages 60–71. Springer, 1999.

[6] P. Brooke. *A Timed Semantics for a Hierarchical Design Notation*. PhD thesis, University of York, 1999.

[7] S. D. Brookes, A. W. Roscoe, and D. J. Walker. An Operational Semantics for CSP. Technical report, 1986.

[8] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Convergence Testing in Term-Level Bounded Model Checking. In *Proc. of the 12th IFIP WG 10.5 Advanced Research Working Conf. on Correct Hardware Design and Veri. Methods (CHARME 2003)*, pages 348–362, 2003.

[9] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential Circuit Verification Using Symbolic Model Checking. In *Proc. of the 27th ACM/IEEE Design Automation Conf. (DAC'90)*, pages 46–51, 1990.

[10] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[11] S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/Event-Based Software Model Checking. In *Proc. of the 4th Inter. Conf. on Integrated Formal Methods (IFM 2004)*, pages 128–147, 2004.

[12] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. of the 14th Inter. Conf. on Computer Aided Veri. (CAV 2002)*, pages 359–364, 2002.

[13] E. M. Clarke, A. Biere, R. Raimi, and Y. S. Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[14] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.

[15] J. S. Dong, P. Hao, S. C. Qin, J. Sun, and Y. Wang. Timed Patterns: TCOZ to Timed Automata. In *Proc. of the 6th Inter. Conf. on Formal Engineering Methods (ICFEM 2004)*, pages 483–498. Springer, 2004.

[16] J. S. Dong, P. Hao, J. Sun, and X. Zhang. A Reasoning Method for Timed CSP Based on Constraint Solving. In *Proc. of the 8th Inter. Conf. on Formal Engineering Methods (ICFEM 2006)*, pages 342–359. Springer, 2006.

[17] J. S. Dong, Y. Liu, J. Sun, and X. Zhang. Verification of Computation Orchestration Via Timed Automata. In *Proc. of the 8th Inte. Conference on Formal Engineering Methods (ICFEM 2006)*, 2006.

[18] P. Gastin and D. Oddoux. Fast LTL to Büchi Automata Translation. In *Proc. of the 13th Inter. Conf. on Computer Aided Verification (CAV 2001)*, pages 53–65. Springer, 2001.

[19] C.A.R. Hoare. *Communicating Sequential Processes*. Inter. Series in Computer Science. Prentice-Hall, 1985. New version at www.usingcsp.com/cspbook.pdf.

[20] G. J. Holzmann. The Model Checker SPIN. *IEEE Trans. on Soft. Eng.*, 23(5):279–295, 1997.

[21] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[22] O. Strichman. Accelerating Bounded Model Checking of Safety Properties. *Formal Methods in System Design*, 24(1):5–24, 2004.

[23] J. Sun, Y. Liu, and J. S. Dong. A Simulator and Model Checker for CSP. http://www.comp.nus.edu.sg/~liuyang/pat/, 2007.

[24] Wenhui Zhang. SAT-Based Verification of LTL Formulas. In *Proc. of the 11th International Workshop FMICS 2006*, pages 277–292, 2006.