Model Checking of Control-Finite CSP Programs

Kenichi Asai Satoshi Matsuoka Akinori Yonezawa

Department of Information Science, Faculty of Science,
The University of Tokyo
7-3-1 Hongou, Bunkyo-Ku, Tokyo, 113, Japan
{asai,matsu,yonezawa}@is.s.u-tokyo.ac.jp

Abstract

We investigate an automatic verification mechanism for control-finite CSP programs using the model checking technique. CSP programs with variables and usual sequential constructs are transformed into transition systems to apply the model checking technique. More specifically, binding of variables are attached to states and sequential statements are treated uniformly as events in a parallel environment. With this transformation, our system can be viewed as implementing the model checker for a static version of value-passing CCS. Another characteristic of the system is that the CSP programs to be verified are not limited to finite-state, but control-finite. CSP programs are control-finite if their control flow is finite. We hope it can serve as one step toward verification of infinite state programs.

1 Introduction

Writing concurrent programs is difficult because they exhibit various unexpected behaviors, such as deadlocks or race-conditions. Without some mechanisms to check their safety, it would be significantly difficult to construct large concurrent programs.

Conventionally, parallel programs have been verified using the *axiomatic approach*[11, 10]: each program construct has an *axiom* or an *inference rule* which describes its meaning. Programs are correct if their proof can be composed with these axioms and rules. Since the axioms include the rule of consequence, that is, all facts that are mathematically provable can be used as axioms, we can deduce a wide range of properties of concurrent programs. The axiomatic proof technique, however, requires deep insight into program behaviors. For example, to verify a program which contains a **while** statement, we have to find a *loop invariant* which holds throughout the statement.

Model checking[4, 8] is a different approach to verifying parallel programs; it can automatically check if a given state of a finite transition system is a model of a given formula, i.e., it checks whether the formula is true in that state. By using formulas in the *modal mu-calculus*[13] to express properties of parallel programs, model checking allows us to check a variety of properties, including absence of deadlock and mutual exclusiveness. The key to model checking is the use of *transition systems*, which describe the external behavior of processes. The model checker verifies the correctness of a parallel program by traversing all possible states over a given transition system that describes the behavior of the program.

The problem, however, is as follows: although transition systems give good abstractions of parallel processes, they are not suited for practical programs. There, we use variables to distinguish different states, **if** statements to cause branches in the control, and **while** statements for repetition. All these features are abstracted away in the transition systems as internal actions.

We present a method for automatically verifying realistic parallel programs which contain sequential constructs. As a first step, we use Hoare's Communicating Sequential Processes (CSP)[9] as a working example. We hope to extend our approach to be able to handle more flexible language based on concurrent objects, such as Actors[1] or ABCL[17].

CSP processes are literally sequential processes which communicate synchronously. CSP includes variables, alternation and repetition constructs, sequential composition, and parallel composition. Verification of a CSP program is performed by first transforming it into a transition system. A state of the transition system consists of *execution pointers* together with a binding of variables. Transition relations are constructed from the control flow of the CSP programs. Then, we apply the model checking technique to the resulting transition system. Verification proceeds fully automatically — we just supply the formula to be verified. With this transformation, our system can be viewed as implementing the model checker for a static version of

$skip:$	**skip**
$assignment:$	$x_1, \ldots, x_n := e_1, \ldots, e_n$ *or* $\overline{x} := \overline{e}$
$send:$	**send** $t(\overline{e})$ **to** p
$receive:$	**receive** $t(\overline{x})$ **from** p

Figure 1: **Simple commands (t: template, p: process name)**

$sequence:$	$c_1; \ldots; c_n$
$parallel:$	$[\, p_1 :: c_1 \,\|\, \ldots \,\|\, p_n :: c_n]$
$alternation:$	**if** $g_1 \rightarrow c_1 \,[]\, \ldots \,[]\, g_n \rightarrow c_n$ **fi**
$repetition:$	**do** $g_1 \rightarrow c_1 \,[]\, \ldots \,[]\, g_n \rightarrow c_n$ **od**

Figure 2: **Composite commands (c_i: command, p_i: process name, g_i: guard)**

value-passing CCS.

Unfortunately, not all CSP programs can be verified with this method because the technique can be applied only to *finite* transition systems. The number of states of CSP programs can easily become infinite, because the range of variable values is unbounded. This means that only programs which contain variables with finite ranges can be checked. To weaken this restriction, we introduce the *lazy evaluation mechanism*, which defers the evaluation of variables until they are required. With this mechanism, we can verify *control-finite* programs. CSP programs are control-finite if their control flow is finite.

We have implemented the model checker in MIT-Scheme, and verified various examples of parallel programming, including the dining philosophers problem and the mutual exclusion problem. Experimental results show that the proofs of these examples complete within a reasonable time.

2 Background

2.1 Overview of CSP

Communicating Sequential Processes (CSP) was introduced by Hoare[9] to serve as a basis for parallel programming languages. The commands of CSP are shown in Figure 1 and 2. The notation $\overline{x}$ represents the vector $x_1, \ldots, x_n$. The communication commands **send** and **receive** were originally written as $p!t(\overline{e})$ and $p?t(\overline{x})$. We use a Pascal-like style for better readability. The guard g_i is either a send command, a receive command, or a skip command, followed by **when** b, where b is a boolean expression (e.g., **send** $t(\overline{e})$ **to** p **when** b, **receive** $t(\overline{x})$ **from** p **when** b, or **skip when** b.) Note that the CSP programs are static. The syntax of CSP programs prevents creation of an infinite number of processes.

2.2 Labeled transition system

A labeled transition system is a simple model for parallel programs. It regards the execution of programs as transitions between states. A labeled transition system is formally defined as follows:

Definition 1 *A labeled transition system is a triple $\mathcal{T} = (\mathcal{S}, \mathcal{L}, \rightarrow)$ where $\mathcal{S}$ is a non-empty set of states, $\mathcal{L}$ is a non-empty set of actions, and $\rightarrow (\subset \mathcal{S} \times \mathcal{L} \times \mathcal{S})$ is a ternary relation representing labeled transitions.*

A labeled transition system is finite if $\mathcal{S}$ is finite. The notation $s_1 \xrightarrow{a} s_2$ is used to mean $(s_1, a, s_2) \in \rightarrow$.

2.3 The modal mu-calculus

The modal mu-calculus is a member of the family of branching-time temporal logic[13]. It can express a variety of properties, including absence of deadlock and mutual exclusiveness. The syntax of the modal mu-calculus formula A is defined by:

$$A ::= Q \mid \neg A \mid A \wedge A \mid [K]A \mid Z \mid \nu Z.A.$$

A formula is either an atomic proposition Q, a negation formula $\neg A$, a conjunction formula $A_1 \wedge A_2$, a *modalized* formula $[K]A$ where K is a set of actions, a propositional variable Z, or a maximal fixed point formula $\nu Z.A$.

The formulas in the modal mu-calculus are interpreted on labeled transition systems with the help of *valuation* V. The valuation is used to interpret atomic propositions and propositional variables. A set of states $||A||_V$ that satisfy the formula A on a transition system $(\mathcal{S}, \mathcal{L}, \rightarrow)$ is inductively defined as follows:

$$||Q||_V = V(Q),$$

$$||\neg A||_V = \mathcal{S} - ||A||_V,$$

$$||A \wedge B||_V = ||A||_V \cap ||B||_V,$$

$$||[K]A||_V = \{s \in \mathcal{S} \mid \forall s'.\forall a \in K. \; if\ s \xrightarrow{a} s'\ then\ s' \in ||A||_V\},$$

$$||Z||_V = V(Z),$$

$$||\nu Z.A||_V = \bigcup \{S' \subseteq S \mid S' = ||A||_{V[Z:=S']}\}.$$

An atomic proposition Q describes an atomic property of states. It is used to distinguish two states with different natures. A negation formula and a conjunction formula need few comments. They express usual *not* and *and*. We will also use *or*, $A_1 \vee A_2$, defined as $\neg(\neg A_1 \wedge \neg A_2)$.

The meaning of a modalized formula $[K]A$ is that after the execution of every executable action taken from K, A holds. We omit curly bracket when we enumerate elements of K as $[a, b, c]A$ instead of $[\{a, b, c\}]A$.

An important derived formula of $[K]A$ is $\langle K\rangle A$, which is defined as $\neg[K]\neg A$. $\langle K\rangle A$ expresses that there is an executable action a in K such that after the execution of a, A holds.

The last two formulas Z and $\nu Z.A$ are used to express recursive predicates. Z may appear only in the body A of $\nu Z.A$ as a bounded variable. $\nu Z.A$ is a maximal solution of the equation $Z = A$. One syntactic restriction on $\nu Z.A$ is that every free occurrence of Z in A must be under an even number of negations. This ensures $\nu Z.A$ to be a monotonic function. It is useful to represent a property which *always* holds. For example, the formula $\nu Z.\langle K\rangle Z$ expresses that some actions taken from K are always executable, and $\nu Z.Q \wedge [K]Z$ says that as long as one is performing actions in K, Q is always true.

2.4 Local model checking

The model checking technique is a method to automatically check if a given state of a *finite* transition system is a *model* of a given formula or not. In other words, model checkers check if the formula holds in that state. Among various model checkers, we introduce the *local* model checker by Stirling and Walker[14]. The local model checker is a tableau system which can test whether a state satisfies a formula without global information. Since it uses only necessary information for checking, it can sometimes check formulas on an infinite transition system, when only finite portions are required for checking.

The tableau system is constructed using inverse natural deduction type rules of the form:

$$\frac{s \vdash_\Delta A}{s_1 \vdash_{\Delta_1} A_1 \ldots s_k \vdash_{\Delta_k} A_k}\ side\ condition.$$

$s \vdash_\Delta A$ is called a sequent. The rule is read as: to check if the state s satisfies the formula A under the *definition list* Δ, check if $s_i \vdash_{\Delta_i} A_i$ holds for $i = 1, \ldots, k$. If they all hold, so does $s \vdash_\Delta A$.

A definition list supplies the environment which stores fixed point formulas encountered so far. Each time a fixed point formula $\nu Z.A$ is encountered, we introduce a new *constant* U, add $U = \nu Z.A$ to the definition list, and check $A[Z := U]$. If U is encountered afterwards, it means that Z in the original formula is encountered. Thus, we unroll the fixed point by looking up the definition list. The notation $\Delta \cdot U = A$ is a definition list obtained by adding a definition $U = A$ to Δ. $\Delta(U)$ returns a formula bound to U. Here are the rules:

$$\frac{s \vdash_\Delta \neg\neg A}{s \vdash_\Delta A}, \qquad \frac{s \vdash_\Delta A \wedge B}{s \vdash_\Delta A \;\; s \vdash_\Delta B},$$

$$\frac{s \vdash_\Delta \neg(A \wedge B)}{s \vdash_\Delta \neg A} \text{ or } \frac{s \vdash_\Delta \neg(A \wedge B)}{s \vdash_\Delta \neg B},$$

$$\frac{s \vdash_\Delta [K]A}{s_1 \vdash_\Delta A \ldots s_n \vdash_\Delta A} \quad \begin{array}{l}\{s_1, \ldots, s_n\} = \\ \quad \{s' \mid \exists a \in K.s \xrightarrow{a} s')\},\end{array}$$

$$\frac{s \vdash_\Delta \neg[K]A}{s' \vdash_\Delta \neg A}\ \exists a \in K.s \xrightarrow{a} s',$$

$$\frac{s \vdash_\Delta \nu Z.A}{s \vdash_{\Delta'} U}\ \Delta' \text{ is } \Delta \cdot U = \nu Z.A,$$

$$\frac{s \vdash_\Delta \neg\nu Z.A}{s \vdash_{\Delta'} U}\ \Delta' \text{ is } \Delta \cdot U = \neg\nu Z.A,$$

$$\frac{s \vdash_\Delta U}{s \vdash_\Delta A[Z := U]}\ \mathcal{C} \text{ and } \Delta(U) = \nu Z.A,$$

$$\frac{s \vdash_\Delta U}{s \vdash_\Delta \neg A[Z := \neg U]}\ \mathcal{C} \text{ and } \Delta(U) = \neg\nu Z.A.$$

Condition $\mathcal{C}$, appearing in the side condition on the last two rules, is that: above the current node $s \vdash_\Delta U$, no node of the form $s \vdash_{\Delta'} U$ for some Δ' appears.

Model checking proceeds in a top-down manner. First, write a state and a formula to be checked with the empty definition list as $s \vdash A$. (We omit the definition list when it is empty.) Then, we apply the rules as far as possible. The rule to be applied is determined by the structure of the formula. When no rules are applicable, the leaf nodes are examined. If they are all *true* leaf nodes, then the state s satisfies the formula A. Note that there is a nondeterminism, or or-branching, in the rules for $\neg(A \wedge B)$ and $\neg[K]A$. This means that we have to choose a correct one whose leaf nodes become true. In practice, we test all cases until a true leaf node is found.

A leaf node $s \vdash_\Delta A$ is true when one of the following requirements holds: (1)$A = Q$ and $s \in V(Q)$, (2)$A = \neg Q$ and $s \notin V(Q)$, (3)$A = [K]B$ for some B, (4)$A = U$ and $\Delta(U) = \nu Z.B$. The case (1) and (2) are clear. $s \vdash_\Delta Q$ and $s \vdash_\Delta \neg Q$ hold if and only if $s \in V(Q)$ and $s \notin V(Q)$, respectively. $s \vdash_\Delta [K]B$ is true because s can perform no actions in K. If it can, the node can not be a leaf node because the rule for modalized formulas is still applicable. The last case needs some comments. $s \vdash_\Delta U$ becomes a leaf node only when the condition $\mathcal{C}$ fails to hold, namely, a node of the form $s \vdash_{\Delta'} U$ has already appeared. Intuitively speaking, as the truth of $s \vdash U$ depends on $s \vdash U$ itself, the fixed point is reached. In this case, it is proved [14] that s satisfies $\nu Z.B$.

The following theorems [14] guarantee that the model checker works correctly.

Theorem 1 *Every tableau for* $s \vdash A$ *on a finite transition system is finite.*

Theorem 2 $s \vdash A$ *holds if and only if* $s \in ||A||_V$.

3 CSP programs as transition systems

In this section, CSP programs are transformed into transition systems to apply the model checking technique. Although CSP programs have internal states and sequential constructs, they can be regarded as transition systems through (1) including a binding of variables into the definition of states, and (2) observing sequential constructs uniformly as actions. We define states, actions, and transition relations for CSP programs in the following sections to obtain transition systems representing CSP programs. Then, we illustrate how the model checking proceeds with some examples.

3.1 States

We define a state of a CSP program as a pair of *execution pointers* and a binding of variables. To determine states of CSP programs, we need two pieces of information. One is where in the program each process is running. It is represented as (a set of) execution pointers or labels attached to appropriate place in the program. The other is a binding of variables. Even though the execution pointers are the same, succeeding behavior may change if the value of variables differ. Thus, we have to include the binding of variables into the definition of states. Assuming that all variables are global, we maintain one binding list for the whole program instead of each process. Local variables are realized by using unique variable names.

Execution pointers indicate the location in the program where the processes are running. A single process is described by a single execution pointer, while a set of processes is described by a set of execution pointers. Execution pointers are attached to the beginning of the program, the end of the program, all ";"s which appear in sequence commands, and all " $\rightarrow$"s which appear in alternative and repetitive commands. In the case where parallel commands appear, we also label the beginning and the end of each processes in parallel commands. Then, the execution pointers before and after the parallel commands are identified with a set of the first execution pointers and the last execution pointers of component processes, respectively. Note that we take simple commands as atomic actions.

See, for example, the program in Figure 3, which sends 5 or 6 to $user$, depending on which of the processes, $proc1$ or $proc2$, is executed first. All ";", as well as the beginning and the end of each process are labeled with execution pointers. ep and ep' will not be used as they are identified with $\{ep_1, ep_3\}$ and $\{ep_2, ep_4\}$, respectively. Figure 4 shows a program of a bounded buffer of capacity one. " $\rightarrow$" in the repetitive command is labeled with ep_1. Ep_1 expresses that $put(x)$ has just been received and $get(x)$ is about to be sent.

$main ::_{ep_0}$ $x := 2;_{ep}$
$[\ proc1 ::_{ep_1} x := x + 1_{ep_2}$
$||\ proc2 ::_{ep_3} x := x * 2_{ep_4}];_{ep'}$
send $t(x)$ **to** $user_{ep_5}$

Figure 3: $Main$ **sends 5 or 6 to** $user$

$buffer ::_{ep_0}$
do receive $put(x)$ **from** $user$ **when true**
$\rightarrow_{ep_1}$ **send** $get(x)$ **to** $user$
od $_{ep_2}$

Figure 4: **A bounded buffer with capacity 1**

3.2 Actions

In CSP programs, state transitions occur when one of the simple commands is executed or when an internal communication occurs. The actions of CSP programs are defined by:

$$\mathcal{L} = Skip \cup Assignment \cup Send \cup Receive \cup \{\tau\}$$

where τ represents an internal communication. $Skip$ and $Assignment$ are sets of all skip and assignment commands.

$Send$ is a set of send commands whose actual parameters are substituted with fresh variables $\overline{u}$. For example, the program

$$p :: \textbf{send } t(5) \textbf{ to } user$$

can perform an action **send** $t(u)$ **to** $user$. The assignment $u := 5$ is performed and we can observe the communicated value through the value of u. The program satisfies the formula $\langle$**send** $t(u)$ **to** $user\rangle u = 5$, which states that the program can take a send action with the sent value 5. Introducing new variables $\overline{u}$ enables us to reference sent values freely.

Likewise, $Receive$ is a set of receive commands whose virtual parameters are substituted with (all possible) actual parameters. The reason for the substitution is fundamental in $Receive$. Consider the program:

$$
\begin{array}{rl}
zerop :: & \textbf{receive } t(n) \textbf{ from } user; \\
& \textbf{if send } yes() \textbf{ to } user \textbf{ when } n = 0 \rightarrow \textbf{skip} \\
& [\,] \textbf{ send } no() \textbf{ to } user \textbf{ when } n \neq 0 \rightarrow \textbf{skip} \\
& \textbf{fi}
\end{array}
$$

which sends yes if the received value is zero and no otherwise. If we allow virtual parameters for receive actions, we can write a formula such as $\langle \textbf{receive } t(n) \textbf{ from } user\rangle\langle \textbf{send } yes() \textbf{ to } user\rangle true$, meaning that after receiving n from user, it can always answer yes. The truth of such formulas can not be determined in a simple way until the value of virtual parameters are known. One way to proceed without knowing the value is to use a symbolic computation mechanism which we discuss in Section 4.3. For now, we forbid the use of virtual parameters in receive actions.

3.3 Transition relations

Transition relations are defined for each action a in $\mathcal{L}$. First, we consider the case where a is not τ. Let ep and ep' be execution pointers just before and after the command in the program designated by the action a, and $(EP, \mathcal{E})$ be a global state where EP is a set of execution pointers and $\mathcal{E}$ is a binding of variables. Assume that the **when** clause of a (if exists) is true and $ep \in EP$, that is, the action a is ready to be taken. Then, the transition relation for a is defined as follows, depending on the type of a:

$$
\begin{array}{l}
(EP, \mathcal{E}) \xrightarrow{\quad\textbf{skip}\quad} (EP', \mathcal{E}), \\
(EP, \mathcal{E}) \xrightarrow{\quad\overline{x} := \overline{e}\quad} (EP', \mathcal{E}[\overline{x} := \overline{e}]), \\
(EP, \mathcal{E}) \xrightarrow{\quad\textbf{send } t(\overline{u}) \textbf{ to } p\quad} (EP', \mathcal{E}[\overline{u} := \overline{e}]), \\
(EP, \mathcal{E}) \xrightarrow{\quad\textbf{receive } t(\overline{d}) \textbf{ from } p\quad} (EP', \mathcal{E}[\overline{x} := \overline{d}])
\end{array}
$$

where EP' is $EP \cup \{ep'\} - \{ep\}$, $\overline{u}$ and $\overline{d}$ are virtual and actual parameters explained in the previous section.

For all cases above, an action proceeds the execution pointer leaving some side effects in $\mathcal{E}$. A skip action leaves nothing. An assignment action changes the binding of variables as just assigned. A send action introduces a binding for new variables $\overline{u}$ so that we can check the sent values. A receive action assigns actual parameters to the virtual parameters.

The transition relation for τ is almost identical except that two processes participate. Consider a typical matching pair:

$$
\begin{array}{l}
p_1 :: \cdots ;_{ep_1} \textbf{ send } t(\overline{e}) \textbf{ to } p_2 \ (\textbf{when } b_1);_{ep'_1} \cdots, \\
p_2 :: \cdots ;_{ep_2} \textbf{ receive } t(\overline{x}) \textbf{ from } p_1 \ (\textbf{when } b_2);_{ep'_2} \cdots .
\end{array}
$$

Let $(EP, \mathcal{E})$ be a global state and assume $ep_1 \in EP$ and $ep_2 \in EP$. If both b_1 and b_2 (if exist) are true in $\mathcal{E}$, the transition relation for τ is:

$$(EP, \mathcal{E}) \xrightarrow{\tau} (EP', \mathcal{E}[\overline{x} := \overline{e}])$$

where EP' is $EP \cup \{ep'_1, ep'_2\} - \{ep_1, ep_2\}$. A τ action is effectively the same as an assignment action $\overline{x} := \overline{e}$ except that it advances both execution pointers.

3.4 Examples

Since we have defined states, actions, and transition relations for CSP programs, we can now apply the model checking algorithm to them. In this section, we demonstrate with two examples how CSP programs are actually transformed into transition systems and see how the model checking algorithm is applied.

The first example illustrates how modalized formulas check all the possible execution paths. We show that the program in Figure 3 actually sends 5 or 6 to $user$. The property is expressed as:

$$
\begin{array}{rl}
Send5or6 = & [Assignment][Assignment][Assignment] \\
& \langle \textbf{send } t(u) \textbf{ to } user\rangle u = 5 \vee u = 6.
\end{array}
$$

There are six possible execution pointers: $ep_0, \{ep_1, ep_3\}, \{ep_2, ep_3\}, \{ep_1, ep_4\}, \{ep_2, ep_4\}$, and ep_5. Actions are defined as $\mathcal{L} = Assignment \cup \{\textbf{send } t(u) \textbf{ to } user\}$, where $Assignment$ is $\{x := 2, x := x + 1, x := x * 2\}$. Here, a new variable u is introduced. Transition relations are given by:

$$
\begin{array}{l}
(ep_0, \mathcal{E}) \xrightarrow{x := 2} (\{ep_1, ep_3\}, \mathcal{E}[x := 2]), \\
(\{ep_1, ep_3\}, \mathcal{E}) \xrightarrow{x := x + 1} (\{ep_2, ep_3\}, \mathcal{E}[x := x + 1]), \\
(\{ep_1, ep_4\}, \mathcal{E}) \xrightarrow{x := x + 1} (\{ep_2, ep_4\}, \mathcal{E}[x := x + 1]), \\
(\{ep_1, ep_3\}, \mathcal{E}) \xrightarrow{x := x * 2} (\{ep_1, ep_4\}, \mathcal{E}[x := x * 2]), \\
(\{ep_2, ep_3\}, \mathcal{E}) \xrightarrow{x := x * 2} (\{ep_2, ep_4\}, \mathcal{E}[x := x * 2]), \\
(\{ep_2, ep_4\}, \mathcal{E}) \xrightarrow{\textbf{send } t(u) \textbf{ to } user} (ep_5, \mathcal{E}[u := x]).
\end{array}
$$

The tableau for the formula is shown in Figure 5(a) and bindings of variables are listed in Figure 5(b). In Figure 5(a), we use two abbreviations: $[A]$ stands for $[Assignment]$ and $\langle\textbf{send}\rangle$ stands for $\langle \textbf{send } t(u) \textbf{ to } user\rangle$. Blanks in the table mean that the value is undefined. Definition lists are omitted because they are not used in this example. In our implemented system, the model checking is done as a depth first search.

At the second line of the tableau, division into two subtableaus occurs, which corresponds to the two possible execution paths. In the last line of subtableaus, $u = 5$ and

$$\dfrac{\dfrac{(ep_0, \mathcal{E}_0) \vdash [A][A][A]\langle \textsf{send} \rangle u = 5 \lor u = 6}{(\{ep_1, ep_3\}, \mathcal{E}1) \vdash [A][A]\langle \textsf{send} \rangle u = 5 \lor u = 6}}{\begin{array}{cc} \dfrac{(\{ep_2, ep_3\}, \mathcal{E}_2) \vdash [A]\langle \textsf{send} \rangle u = 5 \lor u = 6}{\dfrac{(\{ep_2, ep_4\}, \mathcal{E}_3) \vdash \langle \textsf{send} \rangle u = 5 \lor u = 6}{\dfrac{(ep_5, \mathcal{E}_4) \vdash u = 5 \lor u = 6}{\begin{array}{c}(ep_5, \mathcal{E}_4) \vdash u = 6 \\ \text{(success)}\end{array}}}} & \dfrac{(\{ep_1, ep_4\}, \mathcal{E}_5) \vdash [A]\langle \textsf{send} \rangle u = 5 \lor u = 6}{\dfrac{(\{ep_2, ep_4\}, \mathcal{E}_6) \vdash \langle \textsf{send} \rangle u = 5 \lor u = 6}{\dfrac{(ep_5, \mathcal{E}_7) \vdash u = 5 \lor u = 6}{\begin{array}{c}(ep_5, \mathcal{E}_7) \vdash u = 5 \\ \text{(success)}\end{array}}}} \end{array}}$$

Figure 5: **The example proof for a modalized formula (a) the tableau**

	$\mathcal{E}_0$	$\mathcal{E}_1$	$\mathcal{E}_2$	$\mathcal{E}_3$	$\mathcal{E}_4$	$\mathcal{E}_5$	$\mathcal{E}_6$	$\mathcal{E}_7$
x		2	3	6	6	4	5	5
u					6			5

Figure 5: **(b) bindings of variables**

$u = 6$ are nondeterministically chosen from $u = 5 \lor u = 6$. Because the both leaf nodes are true ($\mathcal{E}_4$ satisfies $u = 6$ and $\mathcal{E}_7$ satisfies $u = 5$), we verify that $Send5or6$ holds at the beginning of the program.

The next example illustrates the use of recursive formulas and definition lists. We check if the program in Figure 4 can take two actions **receive** $put(3)$ **from** $user$; **send** $get(u)$ **to** $user$ successively infinite number of times. The property is expressed as: $\nu Z.\langle \textsf{receive} \rangle \langle \textsf{send} \rangle Z$. Here, we write $\langle \textsf{receive} \rangle$ for $\langle \textsf{receive}\ put(3)\ \textsf{from}\ user \rangle$ and $\langle \textsf{send} \rangle$ for $\langle \textsf{send}\ get(u)\ \textsf{to}\ user \rangle$.

There are only two possible (reachable) execution pointers: ep_0 and ep_1. Actions are given by $\mathcal{L} = \{\textsf{receive}\ put(e)\ \textsf{from}\ user, \textsf{send}\ get(u)\ \textsf{to}\ user\}$ where e and u are newly introduced actual and virtual parameters. Note that $\mathcal{L}$ contains infinite number of actions. The actual parameter e must be a concrete value when the receive action is taken. Transition relations are given by:

$$(ep_0, \mathcal{E}) \xrightarrow{\textsf{receive}\ put(e)\ \textsf{from}\ user} (ep_1, \mathcal{E}[x := e]),$$
$$(ep_1, \mathcal{E}) \xrightarrow{\textsf{send}\ get(u)\ \textsf{to}\ user} (ep_0, \mathcal{E}[u := x]).$$

The proof is in Figure 6. At the second line of Figure 6(a), a constant U is introduced. It is used to unroll the recursive formula for two times at lines three and six. The proof can not stop at line five, because the condition $\mathcal{C}$ does not hold. Although the similar sequent appears in the second line, bindings of the variables are not the same. Thus, we proceed the proof until we find exactly the same sequent at line eight.

$$\dfrac{(ep_0, \mathcal{E}_0) \vdash \nu Z.\langle \textsf{receive} \rangle \langle \textsf{send} \rangle Z}{\dfrac{(ep_0, \mathcal{E}_0) \vdash_\Delta U}{\dfrac{(ep_0, \mathcal{E}_0) \vdash_\Delta \langle \textsf{receive} \rangle \langle \textsf{send} \rangle U}{\dfrac{(ep_1, \mathcal{E}_1) \vdash_\Delta \langle \textsf{send} \rangle U}{\dfrac{(ep_0, \mathcal{E}_2) \vdash_\Delta U}{\dfrac{(ep_0, \mathcal{E}_2) \vdash_\Delta \langle \textsf{receive} \rangle \langle \textsf{send} \rangle U}{\dfrac{(ep_1, \mathcal{E}_2) \vdash_\Delta \langle \textsf{send} \rangle U}{\begin{array}{c}(ep_0, \mathcal{E}_2) \vdash_\Delta U \\ \text{(success)}\end{array}}}}}}}}$$

$$\Delta(U) = \nu Z.\langle \textsf{receive} \rangle \langle \textsf{send} \rangle Z$$

(a)

	$\mathcal{E}_0$	$\mathcal{E}_1$	$\mathcal{E}_2$
x		3	3
u			3

(b)

Figure 6: **The example proof for a recursive formula (a) the tableau (b) bindings of variables**

4 Improving the algorithm

In this section, we discuss some techniques to improve the algorithm. The first two are concerned with performance while the later ones are concerned with extending the range of programs which can be verified.

4.1 Use of assertion database

Because the local model checker uses only local information, it sometimes exhibits very inefficient behavior. For example, consider the following program which does nothing but **skip**:

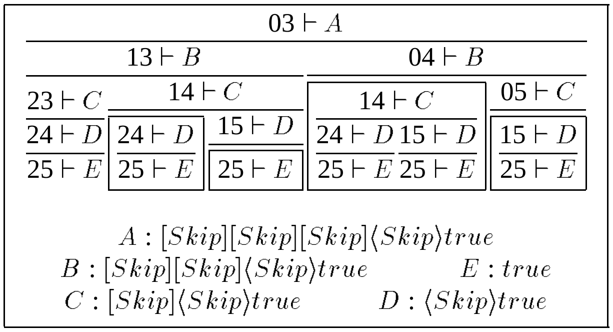

Figure 7: An inefficient proof

$$\begin{array}{ll} main :: & [\ proc1 ::_{ep_0} \textsf{skip}\ ;_{ep_1} \textsf{skip}\ {}_{ep_2} \\ & \parallel proc2 ::_{ep_3} \textsf{skip}\ ;_{ep_4} \textsf{skip}\ {}_{ep_5}]. \end{array}$$

Figure 7 shows a proof tableau for $[Skip][Skip][Skip]\langle Skip\rangle true$. (We write mn for $(\{ep_m, ep_n\},\ \mathcal{E})$.) The tableau is inefficient in that there are repeated occurrences of identical subtableaus. Assuming that model checker checks the left branch in the figure first, the boxed part in the tableau is unnecessary because their truthhood has already been determined.

To avoid this re-evaluation, we introduce an assertion database which stores the known results. The effect of the database is especially significant in verifying CSP programs which contain many independent commands. All skip commands are independent of all other commands in the program. Assignment commands are independent if they do not refer to shared variables. Because the execution order of independent commands does not affect the resulting state of programs, they always reach the same sequent and cause re-evaluation. The experimental results of the effect of databases are presented in Section 5.1.

One drawback of the use of databases is that it suffers from the state explosion problem. Because we record all results calculated so far, the database may become too large to manage. As formulas that we want to verify usually require checking almost all the reachable states, the problem is likely to occur if programs become bigger. Clarke employed Binary Decision Diagrams (BDD) [2] in his (global) model checker, but it can not be directly used for our case because the global state space is unknown. Further study is required here.

4.2 Relation optimization

The purpose of relation optimization is to reduce redundant states. This is because redundant states extremely increase the verification time. For example, consider the program:

$$alwaysSkip ::_{ep_0} \textsf{do skip when true} \rightarrow_{ep_1} \textsf{skip od}$$

which is transformed into:

$$(ep_0, \mathcal{E}) \xrightarrow{\quad\textsf{skip}\quad} (ep_1, \mathcal{E}),$$
$$(ep_1, \mathcal{E}) \xrightarrow{\quad\textsf{skip}\quad} (ep_0, \mathcal{E}).$$

Although $alwaysSkip$ does nothing, it doubles the number of states when it is contained in the program to be verified. All the states are divided into two states depending on which state $alwaysSkip$ is in. Thus, the time required for verification is usually doubled because most properties need all the reachable states to be verified.

To reduce the redundant states, we remove all skip commands, except the ones in guards. In a sequential process in CSP programs, the flow of control branches only in the beginning of alternative and repetitive commands. The execution proceeds straightforwardly in other places. If the execution reaches the point just before a skip command (the one without **when** clause), the execution always proceeds to the point after the skip command. This means that if we have a relation of the form

$$(ep, \mathcal{E}) \xrightarrow{\quad\textsf{skip}\quad} (ep', \mathcal{E})$$

where the **skip** is not the one taken from guards, we can safely remove the relation through substituting all the occurrence of ep in other relations by ep'. The program $alwaysSkip$ is optimized into

$$(ep_0, \mathcal{E}) \xrightarrow{\quad\textsf{skip}\quad} (ep_0, \mathcal{E}).$$

4.3 Towards verification of more expressive programs

Because the model checking technique can be applied to only *finite* transition systems, all variable values in CSP programs have to be finite. In this section, we discuss how to weaken this restriction. We introduce a lazy evaluation mechanism and a simple case of symbolic computation. Through adapting these mechanisms, verification of *control-finite* programs becomes possible. A program is control-finite if all variables in the program that can affect the control flow have finite ranges of values. In this case, the state space of the program becomes finite and the proof always terminates.

Lazy evaluation: Consider the program:

$$incX :: \textsf{do skip when true} \rightarrow x := x + 1\ \textsf{od}$$

which increases x by 1 infinite number of times. Suppose, we want to check that **skip** and $x := x + 1$ can be executed one after the other infinitely: $\nu Z.\langle Skip\rangle\langle x := x + 1\rangle Z$.

Although this is clearly true, the proof tableau for this formula does not terminate. This is because the value of x increases each time and will never take the previous value. As the state never returns to any of the previous states, the proofs for fixed point formulas do not terminate. The example shows that we should not include all the variables into states, but some should be ignored. To include only required variables, we use lazy evaluation strategy.

The lazy evaluation strategy defers the evaluation of variables until variable values are actually required. It evaluates only the necessary variables. In our case, we evaluate the variables only when they appear in **when** clauses in guards (or explicitly mentioned in the verifying formula). As the boolean value in a **when** clause determines the subsequent flow of program execution, it needs to be evaluated.

Because the variable with unevaluated value indicates that it does not affect the program execution, we exclude the variable from states. The variable x in the above example is excluded from states since it does not appear in guards. Without x, the proof terminates.

A simple symbolic computation: Because the unused variables do not affect the control flow, we can prove some more general properties about them. Consider the program of a bounded buffer shown in Figure 4. The restriction on virtual parameters in receive commands forbids us to verify formulas such as $\langle \textsf{\textbf{receive}}\ put(x)\ \textsf{\textbf{from}}\ user\rangle\langle \textsf{\textbf{send}}\ get(u)\ \textsf{\textbf{to}}\ user\rangle u = n$. We have to supply a specific value for n. However, the value of n is never evaluated in the program. In such a case, we just want the model checker to carry n as though n is a specific value.

This is achieved by introducing a simple symbolic computation mechanism. The symbolic computation mechanism enables us to manipulate symbols directly. We allow the use of symbols for unused variables. Because we allow symbols only for unused variables, the mechanism does not affect the model checker itself. We can introduce the symbolic computation orthogonal to the model checker.

5 Experimental results

In this section, various experimental results are presented. The model checker is implemented in the Scheme language[12] with a compiler and runs on NeXT with 28 Mbytes of memory.

5.1 The effect of employing assertion database

In this section, we show how the use of an assertion database reduces the verification time. As a benchmark program, we use the program and the formula shown in Figure 8. Without using the database, the model checker checks the formula for ${}_{2n}C_n$ times. This is understood by considering how the n skip commands in $proc1$ are placed in $2n$ places. On the other hand, if we use the database, the number of states that needs to be checked is $(n+1)^2$.

Table 1 shows the verification time for $n = 3, \ldots, 10$. The ratio in the table expresses the execution time ratio normalized to the case $n = 3$. The ideal ratio is computed as ${}_{2n}C_n / {}_6C_3$ for the case using the database and $(n+1)^2/(3+1)^2$ for the case without it.

Without using the database, the verification time grows rapidly as n becomes large. For $n = 10$, it took 17233.5 seconds, which is about 4.8 hours. If we use the database, on the other hand, the verification time is considerably reduced. Even for $n = 10$, the proof ends in ten seconds.

5.2 Mutual exclusion

In this section, we show the results of verifying various mutual exclusion algorithms. Algorithms are taken from [15], in which the algorithms are verified on the Concurrency Workbench[7]. We reformulate the algorithms in CSP and check their correctness.

We verify six algorithms: algorithms due to Peterson, Dekker, Dijkstra, Knuth, Hyman, and Lamport. All the algorithms consist of two processes with some initial assignment commands. They all use shared variables to achieve mutual exclusion.

Mutual exclusiveness is expressed using an atomic proposition **at** ep as follows:

$$\nu Z.(\neg(\textsf{\textbf{at}}\ critical_1 \wedge \textsf{\textbf{at}}\ critical_2) \wedge [\mathcal{L}]Z).$$

at ep is used to indicate if programs are executing *at* that point. It is true if ep is contained in the current execution pointer. $(\textsf{\textbf{at}}\ critical_1 \wedge \textsf{\textbf{at}}\ critical_2)$ says that one process is executing $critical_1$ and the other process is executing $critical_2$, that is, mutual exclusion is violated. The formula as a whole says that such violation never occurs.

Table 2 summarizes the results of the verifications. The algorithms turn out to preserve mutual exclusion except Hyman's algorithm, which is known to be incorrect. Peterson's

$$main :: \quad [\ proc1 :: \overbrace{\textsf{\textbf{skip}}\ ; \ldots ; \textsf{\textbf{skip}}}^{n} \ \|\ proc2 :: \underbrace{\textsf{\textbf{skip}}\ ; \ldots ; \textsf{\textbf{skip}}}_{n}\ ]$$

$$\underbrace{[Skip] \cdots [Skip]}_{2n-1}\langle Skip\rangle true$$

Figure 8: **Benchmark program and formula**

n	Using a database time (sec.)	ratio	ideal ratio	Without a database time (sec.)	ratio	ideal ratio	Speed-up (without/with)
3	0.7	1.0	1.0	0.7	1.0	1.0	1.0
4	1.3	1.9	1.6	3.0	4.3	3.5	2.3
5	2.0	2.9	2.3	13.7	19.6	12.6	6.9
6	3.1	4.4	3.1	56.3	80.4	46.2	18.2
7	4.6	6.6	4.0	236.2	337.4	171.6	51.3
8	5.8	8.3	5.1	993.8	1419.7	643.5	171.3
9	8.0	11.4	6.3	4145.2	5921.7	2431.0	518.2
10	9.8	14.0	7.6	17233.5	24619.3	9237.8	1758.5

Table 1: **Verification time**

	# of states	result	time (sec.)
Peterson	21	true	10.7
Dekker	87	true	59.1
Dijkstra	161	true	131.4
Knuth	109	true	70.4
Hyman	15	false	6.4
Lamport	15	true	6.0

Table 2: **Results and verification time for mutual exclusion algorithms (for two processes)**

algorithm can be verified in about ten seconds because of its simplicity. The number of states for Hyman's algorithm is small because only necessary states are checked. The number of states for Lamport's algorithm is small because only two processes were used. In this case, the algorithm becomes very simple. For all algorithms, verifications terminated in less than a few minutes.

6 Related work and conclusion

Several local model checkers have been proposed. The algorithm used in this paper is due to Stirling and Walker[14]. Cleaveland[6] proposes a slightly different algorithm with some optimization techniques. Independently of us, he mentions the use of database. Winskel[16] proposes a local model checker in the modal nu-calculus, which is the dual of the modal mu-calculus. The local model checking algorithm has been implemented in Concurrency Workbench[7] by Cleaveland et al. The main difference between Concurrency Workbench and our system is that we employ CSP as its target language, which has internal states (variables). Thus, our system can be viewed as implementing the model checker for a static version of value-passing CCS.

Clarke et al.[4] and Emerson-Lei[8] propose another model checking algorithm, which is not local. With the use of BDD[2], Clarke et al. achieve the verification of processes with a considerably large number of states. Because their model checker uses global state information, the re-evaluation of unnecessary sequents will never occur. Thus, we conjecture that the time complexity of their algorithm is roughly the same as ours with the use of databases.

The relation optimization is a technique to reduce redundant states and is applicable to both local and global model checking algorithms. It is similar in essence to compiler code optimization, and improves the performance dramatically. This technique can be also used in the model checking with abstraction by Clarke et al.[5].

The lazy evaluation strategy and the symbolic computation mechanism are proposed to verify control-finite processes. Although they extend the range of verifiable programs, it is far from our satisfaction. We hope they can serve as one step toward verification of infinite state programs.

The fact that CSP processes are static enables us to use execution pointers to indicate processing points. This is impossible if processes are dynamically created as in CCS where recursion permits us to create infinite number of processes. To extend our system to cope with full CCS, we have to employ a stack or similar data structure instead of execution pointers.

As future research, we are planning to enhance the system in three ways. First, we want to investigate a compositional verification technique by adapting the object oriented technique. Since objects are independent of each other, we could somehow control the combinatorial explosion by treating internal actions in other objects independent. Secondly, we have to manage the state explosion problem. Although it seems we can not directly use BDD, the indirect use of BDD could be possible. The compositional method is also promising. Thirdly, we are seeking a method to synthesize proofs[3]. At present, loop constructs are verified by actually executing loops. Through examining the difference of states upon re-entering a loop, we could synthesize the loop invariant. It will improve performance considerably.

Acknowledgements

We are grateful to Makoto Takeyama for his precise comments on theoretical considerations. Very constructive comments from anonymous referees improved this paper in various ways.

References

[1] Agha, G. *ACTORS: A Model of Concurrent Computation in Distributed Systems*, Cambridge: MIT Press (1986).

[2] Bryant, R. E. "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Comput.*, Vol. C-35, No. 8, pp. 677–691 (1986).

[3] Clarke, E. M. "Synthesis of Resource Invariants for Concurrent Programs," *ACM Trans. Prog. Lang. Syst.*, Vol. 2, No. 3, pp. 338–358 (1980).

[4] Clarke, E. M., E. A. Emerson, and A. P. Sistla "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Prog. Lang. Syst.*, Vol. 8, No. 2, pp. 244–263 (1986).

[5] Clarke, E. M., O. Grumberg, and D. E. Long "Model Checking and Abstraction," *Proc. 19th Ann. ACM Symp. of Principles of Prog. Lang.*, pp. 343–354 (1992).

[6] Cleaveland, R. "Tableau-Based Model Checking in the Propositional Mu-Calculus," *Acta Inf.* 27, pp. 725–747 (1990).

[7] Cleaveland, R., J. Parrow, and B. Steffen "The Concurrency Workbench," *Lecture Notes in Computer Science* 407, pp. 24–37 (1989).

[8] Emerson, E. A., and C-L. Lei "Efficient Model Checking in Fragments of the Propositional Mu-Calculus," *Procedings of 1st IEEE Symp. on Logic in Comput. Sci.*, pp. 267–278 (1986).

[9] Hoare, C. A. R. "Communicating Sequential Processes," *Comm. ACM*, Vol. 21, No. 8, pp. 666–677 (1978).

[10] Levin, G. M., and D. Gries "A Proof Technique for Communicating Sequential Processes," *Acta Inf.* 15, pp. 281–302 (1981).

[11] Owicki, S., and D. Gries "An Axiomatic Proof Technique for Parallel Programs I," *Acta Inf.* 6, pp. 319–340 (1976).

[12] Rees, J., and W. Clinger *Revised^3 Report on the Algorithmic Language Scheme*, SIGPLAN NOTICE, Vol. 21, No. 12, December (1986).

[13] Stirling, C. "Temporal Logics for CCS," *Lecture Notes in Computer Science* 354, pp. 660–672 (1987).

[14] Stirling, C., and D. Walker "Local model checking in the modal mu-calculus," *Theor. Comput. Sci.* 89, pp. 161–177 (1991).

[15] Walker, D. "Automated Analysis of Mutual Exclusion Algorithms using CCS," University of Edinburgh Report ECS-LFCS-89-91, August (1989).

[16] Winskel, G. "A note on model checking the modal ν-calculus," *ICALP '89*, *Lecture Notes in Computer Science* 372, pp. 761–771 (1989).

[17] Yonezawa, A. (Ed.) *ABCL: An Object-Oriented Concurrent System*, Cambridge: MIT Press (1990).