CONPASU-tool: A Concurrent Process Analysis Support Tool based on Symbolic Computation

Yoshinao ISOBE [a,1]

[a] *National Institute of Advanced Industrial Science and Technology, Japan*

Abstract. This paper presents an analysis-method of concurrent processes with value-passing which may cause infinite-state systems. The method consists of two steps: sequentialisation and state-reduction. In the sequentialisation, the symbolic transition graph of a given concurrent process is derived by symbolic operational semantics. In the state-reduction, the number of states in the symbolic transition graph is reduced by removing needless internal transitions. Furthermore, this paper introduces an analysis-tool called `CONPASU`, which implements the analysis-method, and demonstrates how `CONPASU` can be used for automatically analyzing concurrent processes. For example, it can extract abstract behaviors, which are useful for understanding complex behaviors, by focusing on some interesting events.

Keywords. symbolic operational semantics, automatic analysis tool, state-reduction, infinite state process, value-passing process algebra

Introduction

Concurrent processes, which consist of communicating component-processes, are needed in parallel computation environments such as multi-core CPU and distributed systems. It is, however, not easy to completely understand the whole behavior of concurrent processes because it is a result of interactions between communicating component-processes. For example, in order to know the causality between events performed at the different component-processes, internal communications between the component-processes must be considered.

Process algebra such as CSP [1,2] and CCS [3] is a formal framework for analyzing concurrent processes. In process algebra, *implementations* and *specifications* of concurrent processes can be formally described, and then equalities and/or refinements between an implementation and a specification can be checked. In general, the implementation is a model to express the structure of the concurrent-process and the behaviors of its component-processes, while the specification is often a sequential process to express the whole behavior of the concurrent-process.

Various tools [4,5,6,7], called *model checker*, based on process algebra have been developed for automatically checking such equalities and/or refinements, when formal descriptions of an implementation and a specification are given. For example, the following expression is a formal description (in CSP) of the concurrent process `IMPL` which consists of two component-processes `IN` and `OUT`.

[1] Corresponding Author: Yoshinao Isobe, Information Technology Research Institute, National Institute of Advanced Industrial Science and Technology, Tsukuba Central 2, 1-1-1, Umezono, Tsukuba, Ibaraki, 305-8568 Japan, E-mail: `y-isobe@aist.go.jp`.

$$\begin{aligned}
\mathtt{IMPL} &= (\mathtt{IN}\ [|\{|\ \mathtt{com}\ |\}|]\ \mathtt{OUT}) \setminus \{|\ \mathtt{com}\ |\} \\
\mathtt{IN} &= \mathtt{in}?x \rightarrow \mathtt{com}!x \rightarrow \mathtt{IN} \\
\mathtt{OUT} &= \mathtt{com}?y \rightarrow \mathtt{out}!y \rightarrow \mathtt{OUT}
\end{aligned}$$

The first line defines the structure of IMPL, and the second and the third lines define behaviors of IN and OUT, respectively. The meaning of symbols such as → is explained in Section 1. Here, we assume to expect that IMPL behaves like a buffer whose capacity is 2, thus the specification can be formally described as follows:

$$\begin{aligned}
\mathtt{SPEC} &= \mathtt{in}?x \rightarrow \mathtt{SPEC1}(x) \\
\mathtt{SPEC1}(y) &= \mathtt{in}?x \rightarrow \mathtt{SPEC2}(x,y) \ \Box\ \mathtt{out}!y \rightarrow \mathtt{SPEC} \\
\mathtt{SPEC2}(x,y) &= \mathtt{out}!y \rightarrow \mathtt{SPEC1}(x)
\end{aligned}$$

In fact, the behaviors of IMPL and SPEC are equal (e.g. failures-equivalent [2]) and the equality can be automatically checked by the model checker FDR [4] if the range of input is finitised.

As shown in the example of IMPL and SPEC, model checker is very useful for checking relations between an implementation and a specification. It is, however, sometimes difficult to formally describe specifications. Implementations such as IMPL are often hierarchical and complex, but they can be more mechanically described than specifications such as SPEC because designs of structures of systems and behaviors of components are usually given while it is often difficult to formally describe expected behaviors of systems. Our analysis-tool CONPASU can automatically generate the specification SPEC from the implementation IMPL.

In this paper, we present an analysis-method for generating specifications (i.e. sequential processes) from implementations of concurrent processes based on CSP (Communicating Sequential Processes) [1,2] with value-passing. The analysis-method consists of two steps: sequentialisation of concurrent processes by symbolic operational semantics and state-reduction by removing needless internal transitions. It is possible to extract abstract behaviors from the whole behavior by focusing on only interesting events. Then, we introduce an analysis-tool CONPASU, which implements the analysis-method, and demonstrate how it analyzes concurrent processes. The analysis-method and CONPASU have the following features:

- Symbolic transition graphs can be often *finite* even for value-passing processes with variables whose ranges are infinite because variables are not instantiated to values.
- Each symbolic transition has assignments (e.g. $n := n+1$) for updating variables and it has a location for indicating which processes participate in the transition.
- The presented state-reduction method can be directly applied to the symbolic transition graphs without instantiating variables.
- The tool CONPASU generates symbolic transition graphs from concurrent processes described in CSP$_M$ used in FDR [4], and then it can *automatically* reduce the number of states in symbolic transition graphs with preserving *stable-failures-equivalence*.

This paper is organised as follows: First, we give a definition of process algebra with symbolic semantics mainly according to [8] in Section 1. It is used for sequentialising concurrent processes. In Section 2, we present an analysis-method for reducing the number of states with preserving stable-failures-equivalence based on symbolic approach. Then, in Sections 3 and 4, we introduce a tool CONPASU which implements the analysis-method presented in this paper, and demonstrate how CONPASU can analyze concurrent processes. Finally, we compare this work with related works.

1. Process Algebra with Symbolic Operational Semantics

The analysis-method presented in this paper can be applied to concurrent processes whose behaviors are expressed by labeled transition systems, without respect to the syntax. It is, however, convenient to use process algebra for expressing concurrent processes. In this section, we briefly introduce a sub-calculus of the process algebra CSP [1,2] in Subsection 1.1 and define the symbolic operational semantics with data-assignment and locality for the sub-calculus in Subsection 1.2.

1.1. Syntax

We assume that the following sets are given: a set $\mathtt{Var}$ of *variable* ranged over by $x, y, \ldots$, a set $\mathtt{Val}$ of *values* ranged over by $v, \ldots$, a set $\mathtt{Dexp}$ of *data-expressions* ranged over by $e, \ldots$, and a set $\mathtt{Bexp}$ of *Boolean-expressions* ranged over by $b, \ldots$, where $\mathtt{Dexp}$ includes $\mathtt{Var} \cup \mathtt{Val}$ and $\mathtt{Bexp}$ includes $\mathtt{Var} \cup \{\mathtt{true}, \mathtt{false}\}$. Furthermore, we also assume that a set $\mathtt{Chan}$ of *channel-names,* ranged over by $c, \ldots$, and a set $\mathtt{PN}$ of *process-names,* ranged over by $A, \ldots$ are given.

Then, the set $\mathtt{Event}$ of *events*, ranged over by $a, \ldots$, is defined as follows:

$$\mathtt{Event} = \{c!e \mid c \in \mathtt{Chan}, e \in \mathtt{Dexp}\} \cup \{c?x \mid c \in \mathtt{Chan}, x \in \mathtt{Var}\}$$

where the event $c!e$ means sending the evaluation result of e to the channel c, and the event $c?x$ means receiving a value from the channel c and x is bound to the value. The set $\mathtt{Event}$ does not contain basic-events which do not pass values and are used just for synchronization. Such basic-events, however, can be expressed by sending a dummy value, for example zero. In this paper, $c!0$ is sometimes abbreviated to c if the value 0 has no meaning.

The language used in this paper is the set $\mathcal{E}$ of *processes*, ranged over by $E, F, \ldots$, and it is a sub-calculus of CSP [1,2] as defined in Definition 1.1.

Definition 1.1 *The syntax of processes E is given by*

$$E \ ::= \mathtt{STOP} \mid a \to E \mid E \,\Box\, E \mid E \sqcap E \mid E\,[|C|]\,E \mid E \setminus C \mid b \,\&\, E \mid A(\tilde{e})$$

where $a \in \mathtt{Event}$, $C \subseteq \mathtt{Chan}$, $b \in \mathtt{Bexp}$, and $A \in \mathtt{PN}$. And $\tilde{e} \in \mathtt{Dexp}^n$ is an abbreviation of n data-expressions $e_1, \ldots, e_n$. $A(\tilde{e})$ is the process obtained from $A(\tilde{x})$ by replacing n arguments $\tilde{x} \in \mathtt{Var}^n$ by $\tilde{e}$. ▮

Since the semantics of processes is given in the next subsection, each operator is *briefly* explained here: $a \to E$ can perform the event a and thereafter behaves like E. $E \,\Box\, F$ and $E \sqcap F$ represent choices between E and F, where the choice of $E \,\Box\, F$ is externally made by an event of either E or F, while the choice of $E \sqcap F$ is internally made. $E\,[|C|]\,F$ represents a concurrent composition of E and F, where they communicate through channels included in the set C and independently perform events whose channel is not in C. $E \setminus C$ hides communications through channels in C. $b \,\&\, E$ behaves like E if b is true, otherwise it is inactive. The operators have decreasing binding power in the following order: $E \setminus C$, $a \to E$, $b \,\&\, E$, $E \,\Box\, F$, $E \sqcap F$, and $E\,[|C|]\,F$.

The sets of *bound variables* and *free variables* in the process $E \in \mathcal{E}$ are denoted by $\mathtt{bv}(E)$ and $\mathtt{fv}(E)$, respectively, where each input event $c?x$ binds the variable x in E of $c?x \to E$. Similarly, the sets of free variables of the data-expression e and the Boolean-expression b are denoted by $\mathtt{fv}(e)$ and $\mathtt{fv}(b)$, respectively. The set of processes which have no free variable is denoted by $\mathcal{P}$ and ranged over by $P, Q, \ldots$.

The meaning of each process-name is given by a defining equation. We assume that for every process-name $A \in \mathtt{PN}$, there is a defining equation of the form $A(\tilde{x}) = E$, where $E \in \mathcal{E}$ and E has no free variables except $\tilde{x} \in \mathtt{Var}^n$. Process-names are often used for expressing recursive behaviors. For example, let the process-name $\mathtt{SQ}(n)$ be defined by

$$\mathtt{SQ}(n) = ((n > 0) \,\&\, \mathtt{in}?x \to \mathtt{sq}!(x * x) \to \mathtt{SQ}(n-1)) \,\Box\, ((n == 0) \,\&\, \mathtt{end} \to \mathtt{STOP}).$$

If n is greater than 0, $\mathtt{SQ}(n)$ firstly receives a value, to which x is bound, from the channel `in`, and then sends $(x * x)$ to the channel `sq`, and thereafter behaves like $\mathtt{SQ}(n-1)$. And if n is 0, $\mathtt{SQ}(n)$ performs the event `end`, which is the abbreviation of `end!0`, and then stops. In other words, $\mathtt{SQ}(n)$ iteratively receives a value and sends the square of the value n-times, and thereafter performs `end` and then stops.

1.2. Symbolic Operational Semantics

In our analysis method, variables are *symbolically* computed without instantiated to values. It means that symbolic approach can express behaviors of processes with value-passing in a *finite* graph even if ranges of variables are not finitised or parameters are not fixed. Symbolic labeled transition systems have been studied, for example in [8,9,10]. In this subsection, we define a symbolic operational semantics of CSP, based on the standard operational semantics of CSP [2] with symbolic semantics of CSP (e.g. [11]), and extended with data-assignment [9] and locality [12]. The locality has been studied in process algebra (e.g. [12,13]) for giving non-interleaving semantics. In this paper, however, we use interleaving semantics, thus locations are ignored when checking equality. The locality is used for checking independence between transitions when reducing states. The symbolic semantics defined in this section is a combination of existing results [2,9,12] and no new technique is used.

At first, a notation for assigning data-expressions to variables is introduced. An *assignment* has the form $(\tilde{x} := \tilde{e})$, which is an abbreviation of $(x_1 := e_1, \ldots, x_n := e_n)$, and means to simultaneously replace every free variable $x_i \in \mathtt{Var}$ by $e_i \in \mathtt{Dexp}$. The set of assignments is denoted by `Assign` and is ranged over by $\theta, \ldots$. The sets of the domain and the free variables of an assignment $(\tilde{x} := \tilde{e})$ are denoted by $\mathtt{dm}(\tilde{x} := \tilde{e}) = \{x_1, \ldots, x_n\}$ and $\mathtt{fv}(\tilde{x} := \tilde{e}) = \mathtt{fv}(\tilde{e})$, respectively. An assignment θ can be applied to processes E, data-expressions e, and Boolean-expressions b, and they are denoted by $E\theta$, $e\theta$, and $b\theta$. For example,

$$(\mathtt{in}?x \to \mathtt{out}!(x+y) \to \mathtt{STOP})(x := 1, y := 2) = (\mathtt{in}?x \to \mathtt{out}!(x+2) \to \mathtt{STOP}).$$

Then, the composition $\theta \circ \theta'$ such that $E(\theta \circ \theta') = (E\theta)\theta'$ of two assignments can be defined by $(\tilde{x} := \tilde{e}) \circ \theta' = (\tilde{x} := \tilde{e}\theta')(\theta' - \tilde{x})$, where $(\tilde{x} := \tilde{e}\theta)$ represents $(x_i := e_i\theta)$ for every i and $(\theta' - \tilde{x})$ is the assignment obtained from θ' by removing the assignments to $\tilde{x}$.

If an assignment θ has no free variable (i.e. $\mathtt{fv}(\theta) = \emptyset$) and its domain is the set of all variables (i.e. $\mathtt{dm}(\theta) = \mathtt{Var}$), then it is called an *evaluation*. We denote the set of evaluations by `Eval` and let $\rho, \ldots$ range over evaluations.

Next, locality is introduced for indicating where events occur in concurrent processes. In the same way to [12], a *location* δ is a binary tree defined by

$$\delta \ ::= \ \mathtt{0} \ \big| \ \mathtt{1} \ \big| \ (\delta\delta)$$

and the set of locations is denoted by `Loc`, where `0` means the inactive location and `1` means the active location. For example, in the concurrent process $(E_0 \, [|C_1|] \, (E_1 \, [|C_2|] \, E_2))$, the location $(\mathtt{0}(\mathtt{10}))$ is attached to events which E_1 independently performs and $(\mathtt{1}(\mathtt{01}))$ is attached to events which both E_0 and E_2 participate in. The locations are used for checking the causality between events (e.g. the locations $(\mathtt{0}(\mathtt{10}))$ and $(\mathtt{1}(\mathtt{01}))$ are independent), when searching for reducible states (see Definition 2.5).

Then, we define the set `Act` of *actions*, which are guarded events with assignments and locations, by

$$\mathtt{Act} = \{\alpha[b]/\theta@\delta \ \mid \ \alpha \in \mathtt{Event} \cup \{\tau\}, \, b \in \mathtt{Bexp}, \, \theta \in \mathtt{Assign}, \delta \in \mathtt{Loc}\},$$

where τ is a special event, called *internal event*, which cannot be observed ($\tau \notin \mathtt{Event}$), and the set $\mathtt{Event} \cup \{\tau\}$ is ranged over by $\alpha, \ldots$. The action $\alpha[b]/\theta@\delta$ means if the condition b is true then the event α can occur at the location δ, and thereafter variables are updated by the

$$\textbf{E.Snd}\ \frac{}{c!e \to E \bullet\!\!\xrightarrow{c!e@1} E} \qquad \textbf{E.Rcv}\ \frac{}{c?x \to E \bullet\!\!\xrightarrow{c?y@1} E(x := y)}\ (y \text{ is fresh})$$

$$\textbf{E.ECh}_1\ \frac{E \bullet\!\!\xrightarrow{\alpha[b]/\theta@\delta} E'}{E \,\Box\, F \bullet\!\!\xrightarrow{\alpha[b]/\theta@\delta} E'}\ (\alpha \neq \tau) \qquad \textbf{E.ECh}_2\ \frac{E \bullet\!\!\xrightarrow{\tau[b]/\theta@\delta} E'}{E \,\Box\, F \bullet\!\!\xrightarrow{\tau[b]/\theta@\delta} E' \,\Box\, F} \qquad \textbf{E.ICh}_1\ \frac{}{E \sqcap F \bullet\!\!\xrightarrow{\tau@1} E}$$

$$\textbf{E.Grd}\ \frac{E \bullet\!\!\xrightarrow{\alpha[b']/\theta@\delta} E'}{b \,\&\, E \bullet\!\!\xrightarrow{\alpha[b \wedge b']/\theta@\delta} E'} \qquad \textbf{E.PN}\ \frac{E(\tilde{x} := \tilde{e}) \bullet\!\!\xrightarrow{\alpha[b]/\theta@\delta} E'}{A(\tilde{e}) \bullet\!\!\xrightarrow{\alpha[b]/\theta@\delta} E'}\ (A(\tilde{x}) = E)$$

$$\textbf{E.Par}_1\ \frac{E \bullet\!\!\xrightarrow{\alpha[b]/\theta@\delta} E'}{E\,[|C|]\,F \bullet\!\!\xrightarrow{\alpha[b]/\theta@(\delta 0)} E'\,[|C|]\,F}\ (\mathtt{ch}(\alpha) \notin C)$$

$$\textbf{E.Par}_2\ \frac{E \bullet\!\!\xrightarrow{c!e[b]/\theta@\delta} E' \qquad F \bullet\!\!\xrightarrow{c?x[b']/\theta'@\delta'} F'}{E\,[|C|]\,F \bullet\!\!\xrightarrow{c!e[b \wedge b']/\theta\theta'(x:=e)@(\delta\delta')} E'\,[|C|]\,F'}\ (c \in C)$$

$$\textbf{E.Par}_3\ \frac{E \bullet\!\!\xrightarrow{c!e[b]/\theta@\delta} E' \qquad F \bullet\!\!\xrightarrow{c!e'[b']/\theta'@\delta'} F'}{E\,[|C|]\,F \bullet\!\!\xrightarrow{c!e[(e=e') \wedge b \wedge b']/\theta\theta'@(\delta\delta')} E'\,[|C|]\,F'}\ (c \in C)$$

$$\textbf{E.Par}_4\ \frac{E \bullet\!\!\xrightarrow{c?x[b]/\theta@\delta} E' \qquad F \bullet\!\!\xrightarrow{c?x'[b']/\theta'@\delta'} F'}{E\,[|C|]\,F \bullet\!\!\xrightarrow{c?x[b \wedge b']/\theta\theta'@(\delta\delta')} E'\,[|C|]\,F'(x' := x)}\ (c \in C)$$

$$\textbf{E.Hide}_1\ \frac{E \bullet\!\!\xrightarrow{c!e[b]/\theta@\delta} E'}{E \setminus C \bullet\!\!\xrightarrow{\tau[b]/\theta@\delta} E' \setminus C}\ (c \in C) \qquad \textbf{E.Hide}_2\ \frac{E \bullet\!\!\xrightarrow{c?x[b]/\theta@\delta} E'}{E \setminus C \bullet\!\!\xrightarrow{\tau[b]/\theta(x:=v)@\delta} E' \setminus C}\ (c \in C, v \in \mathtt{Val})$$

$$\textbf{E.Hide}_3\ \frac{E \bullet\!\!\xrightarrow{\alpha[b]/\theta@\delta} E'}{E \setminus C \bullet\!\!\xrightarrow{\alpha[b]/\theta@\delta} E' \setminus C}\ (\mathtt{ch}(\alpha) \notin C)$$

Figure 1. The inference rules for transitions $\bullet\!\!\xrightarrow{\alpha[b]/\theta@\delta}$ for performing events (symmetric rules are omitted)

assignment θ. The true condition $[\mathtt{true}]$, the identical assignment $/\varepsilon$, and the unique-location $@1$ are often omitted like $\alpha/\theta@\delta$, $\alpha[b]@\delta$, and $\alpha[b]/\theta$.

By using the actions as labels, two symbolic transitions are defined.

Definition 1.2 *Two symbolic transitions $\bullet\!\!\longrightarrow \subseteq \mathcal{E} \times \mathtt{Act} \times \mathcal{E}$ and $\rightsquigarrow \subseteq \mathcal{E} \times \mathtt{Assign} \times \mathcal{E}$ are the smallest relations satisfying the inference rules in Figures 1 and 2, respectively. For convenience, we write $E \bullet\!\!\xrightarrow{\alpha[b]/\theta@\delta} E'$ and $E \overset{\theta}{\rightsquigarrow} E'$ for $(E, \alpha[b]/\theta@\delta, E') \in \bullet\!\!\longrightarrow$ and $(E, \theta, E') \in \rightsquigarrow$, respectively.* ∎

The first transition $E \bullet\!\!\xrightarrow{\alpha[b]/\theta@\delta} E'$ is used for performing the event α. The side condition of the rule **E**.**Rcv** means that the bound variable is renamed to a *fresh* name, which is not used in the other processes, in order to avoid conflicting with free variables, if necessary. Though the renaming is represented only in the rule **E**.**Rcv** for simplicity, the bound variables can be actually renamed later, for example when composing processes by **E**.**Par**$_i$. We have implemented such renaming mechanism in the tool CONPASU introduced in Section 3, and for example the following transition can be inferred.

$$\textbf{D.STOP}\,\frac{}{\mathtt{STOP} \overset{\varepsilon}{\rightsquigarrow} \mathtt{STOP}} \qquad \textbf{D.Act}\,\frac{}{\alpha.E \overset{\varepsilon}{\rightsquigarrow} \alpha.E}$$

$$\textbf{D.PN}\,\frac{}{A(\tilde{e}) \overset{\tilde{y}:=\tilde{e}}{\rightsquigarrow} A(\tilde{y})}\,(A(\tilde{x}) = E,\ \tilde{y} \text{ are fresh and distinct})$$

$$\textbf{D.Par}\,\frac{E \overset{\theta_0}{\rightsquigarrow} E' \quad F \overset{\theta_1}{\rightsquigarrow} F'}{E \oplus F \overset{\theta_0\theta_1}{\rightsquigarrow} E' \oplus F'}\,(\oplus \in \{\Box,\ \sqcap,\ [|C|]\ \})$$

$$\textbf{D.Res}\,\frac{E \overset{\theta}{\rightsquigarrow} E'}{E\backslash C \overset{\theta}{\rightsquigarrow} E'\backslash C} \qquad \textbf{D.Grd}\,\frac{E \overset{\theta}{\rightsquigarrow} E'}{b \,\&\, E \overset{\theta}{\rightsquigarrow} b \,\&\, E'}$$

Figure 2. The inference rules for transitions $\overset{\theta}{\rightsquigarrow}$ for updating data

$$(\mathtt{in}?x \to P(x))\,[|\emptyset|]\,(\mathtt{out}!x \to Q(x)) \;\bullet\!\!\xrightarrow{\mathtt{in}?x_0@(\mathtt{10})}\; P(x_0)\,[|\emptyset|]\,(\mathtt{out}!x \to Q(x))$$

By renaming x to x_0, the value received through the channel `in` is correctly passed to $P(x_0)$ and not to $Q(x)$. It is also noted that the location $(\mathtt{10})$ means that the left process performs the event. If two or more processes synchronise, then all the locations of the processes are indicated by the active symbol `1`, for example the following transition can be inferred.[1]

$$(\mathtt{com}!x \to P(x))\,[|\{|\ \mathtt{com}\ |\}|]\,(\mathtt{com}?y \to Q(y)) \;\bullet\!\!\xrightarrow{\mathtt{com}!x/(y:=x)@(\mathtt{11})}\; P(x)\,[|\{|\ \mathtt{com}\ |\}|]\,Q(y)$$

The second transition $E \overset{\theta}{\rightsquigarrow} E'$ is used for updating variables by the assignment θ when process-names are unfolded. For example, the transitions from the process $\mathtt{A}(n)$ defined by $\mathtt{A}(n) = \mathrm{up}!n \to \mathtt{A}(n+1)$ are inferred by the rules in Figures 1 and 2 as follows:

$$\mathtt{A}(n) \;\bullet\!\!\xrightarrow{\mathrm{up}!n}\; \mathtt{A}(n+1) \overset{(n:=n+1)}{\rightsquigarrow} \mathtt{A}(n)$$

It means that the transition graph of $\mathtt{A}(n)$ is finite, where the location $@\mathtt{1}$ is omitted. On the other hand, the standard transitions for $\mathtt{A}(n)$ are inferred by the standard operational semantics [2] as follows, when the initial value of n is 0:

$$\mathtt{A}(0) \xrightarrow{\mathrm{up}!0} \mathtt{A}(1) \xrightarrow{\mathrm{up}!1} \mathtt{A}(2) \xrightarrow{\mathrm{up}!2} \mathtt{A}(3) \xrightarrow{\mathrm{up}!3} \cdots.$$

It means that the standard transition graph becomes infinite.

Then, by composing the two symbolic transitions in Definition 1.2, the symbolic operational semantics used in this paper is defined.

Definition 1.3 *The* symbolic operational semantics with assignments and locations *is given by the symbolic labeled transition system* $(\mathcal{E}, \mathtt{Act}, \bullet\!\!\longrightarrow\!\!\!\!\rightarrow\, \subseteq \mathcal{E} \times \mathtt{Act} \times \mathcal{E})$*, where* $\bullet\!\!\longrightarrow\!\!\!\!\rightarrow$ *is defined by*

$$E \;\bullet\!\!\xrightarrow{\alpha[b]/\theta@\delta}\!\!\!\!\rightarrow\; E'' \Leftrightarrow (\exists\, E', \theta_1, \theta_2.\ E \;\bullet\!\!\xrightarrow{\alpha[b]/\theta_1@\delta}\; E',\, E' \overset{\theta_2}{\rightsquigarrow} E'',\, \theta = \theta_2 \circ \theta_1).$$

∎

The process $\mathtt{SQ}(n)$, given in Subsection 1.1, is used again here. Figure 3 shows the transition graph derived from $\mathtt{SQ}(n)$ by the symbolic operational semantics in Definition 1.3. The transition graph shows that $\mathtt{SQ}(n)$ iteratively receives a value and sends the square of the value n-times, and thereafter performs `end` and then stops for any n.

[1] In this paper, we denote the set of channels $c_1, \ldots, c_n$ by $\{|\ c_1, \ldots, c_n\ |\}$ rather than $\{c_1, \ldots, c_n\}$ according to the syntax of CSP$_M$ used in FDR[4].

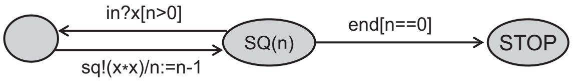


Figure 3. The transition graph of $\mathtt{SQ}(n)$ by the symbolic operational semantics with assignment

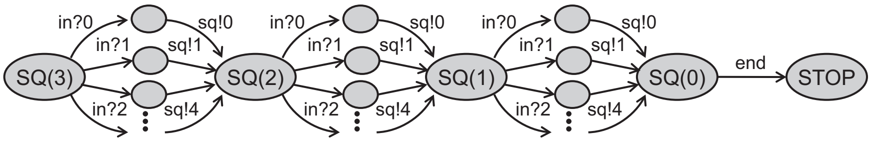


Figure 4. The transition graph of $\mathtt{SQ}(3)$ by the standard operational semantics

To compare the symbolic semantics with standard semantics given in [2], we show the transition graph of $\mathtt{SQ}(n)$ by the standard operational semantics in Figure 4, where every variable has to be instantiated to a value for each transition. The graph in Figure 4 has infinite number of branches because the range of values received through the channel `in` is not restricted, and the initial value of n must be fixed. On the other hand, the graph in Figure 3 is finite for any n because infinite number of values can be expressed by the variable x.

Here, we give the relations between the symbolic operational semantics $\overset{\alpha[b]/\theta@\delta}{\bullet\!\!\twoheadrightarrow}$ and the (not-symbolic) standard operational semantics $\xrightarrow{\alpha_0}$ in [2], where $\alpha_0 \in \mathtt{Act}_0$ and $\mathtt{Act}_0$ is the set of events without variables:

$$\mathtt{Act}_0 = \{c.v \mid c \in \mathtt{Chan}, v \in \mathtt{Val}\} \cup \{\tau\}.$$

It is similar to the result presented for value-passing CCS [8,9].

Lemma 1.1 *$E\rho \xrightarrow{\alpha} P'$ if and only if for some b, θ, δ, and E', either*

$$\begin{array}{ll} & \textit{for some } c, v, x,\ E \overset{c?x[b]/\theta@\delta}{\bullet\!\!\longrightarrow\!\!\!\!\twoheadrightarrow} E',\ \alpha = c.v,\ b\rho, \textit{ and } P' \equiv E'\theta(x := v)\rho, \\ \textit{or} & \textit{for some } c, e,\ E \overset{c!e[b]/\theta@\delta}{\bullet\!\!\longrightarrow\!\!\!\!\twoheadrightarrow} E',\ \alpha = c.(e\rho),\ b\rho, \textit{ and } P' \equiv E'\theta\rho, \\ \textit{or} & E \overset{\tau[b]/\theta@\delta}{\bullet\!\!\longrightarrow\!\!\!\!\twoheadrightarrow} E',\ \alpha = \tau,\ b\rho, \textit{ and } P' \equiv E'\theta\rho. \end{array}$$

∎

Finally, the process-name $\mathtt{SP}_{(\twoheadrightarrow,E)}(\tilde{x})$ is defined for generating a sequential process for *any* symbolic transition relation $\twoheadrightarrow$.

Definition 1.4 *Let $\twoheadrightarrow \subseteq \mathcal{E} \times \mathtt{Act} \times \mathcal{E}$ be a symbolic transition relation. Then, for any process $E \in \mathcal{E}$, the process-name $\mathtt{SP}_{(\twoheadrightarrow,E)}(\tilde{x})$ is defined as follows:*

$$\mathtt{SP}_{(\twoheadrightarrow,E)}(\tilde{x}) = \ \Box\{b \,\&\, \mathtt{obs}(\alpha) \rightarrow (\mathtt{SP}_{(\twoheadrightarrow,E')}(\tilde{x}')\theta) \mid E \overset{\alpha[b]/\theta@\delta}{\longrightarrow\!\!\!\!\twoheadrightarrow} E'\}$$

where $\tilde{x}$ of $\mathtt{SP}_{(\twoheadrightarrow,E)}(\tilde{x})$ is the list of free variables of E, $\ \Box\ \{E_1, \ldots, E_n\}$ is an abbreviation of $E_1 \ \Box \cdots \Box\ E_n$, namely replicated external choice, and $\mathtt{obs}(\alpha)$ is defined as follows:

$$\mathtt{obs}(\alpha) = \begin{cases} \mathtt{tmp} & (\textit{if } \alpha = \tau) \\ \alpha & (\textit{otherwise}) \end{cases}$$

where `tmp` is a special observable event which is not used in the processes E. As a special case, if $\twoheadrightarrow$ is $\bullet\!\!\twoheadrightarrow$, which is defined in Definition 1.3, it is often omitted, thus

$$\mathtt{SP}_{(E)}(\tilde{x}) = \mathtt{SP}_{(\bullet\!\!\longrightarrow\!\!\!\!\twoheadrightarrow,E)}(\tilde{x}).$$

∎

The event tmp is used instead of the internal event τ for choosing one from external choice processes ($E_1 \;\square\; \cdots \;\square\; E_n$) because the external choice is not executed by τ (see the rule $\mathbf{E}.\mathbf{ECh}_2$ in Figure 1). As expected, E and $\mathtt{SP}_{(E)}(\tilde{x})\backslash\{\mathtt{tmp}\}$ are *strongly bisimilar*[3], where locations δ are ignored, because the following relation can be easily proved:

$$\mathtt{SP}_{(E)}(\tilde{x})\backslash\{\mathtt{tmp}\} \overset{\alpha[b]/\theta@\mathtt{1}}{\bullet\!\!\longrightarrow\!\!\!\!\rightarrow} \mathtt{SP}_{(E')}(\tilde{x}')\backslash\{\mathtt{tmp}\} \iff \exists\,\delta.\, E \overset{\alpha[b]/\theta@\delta}{\bullet\!\!\longrightarrow\!\!\!\!\rightarrow} E'.$$

2. State Reduction

As shown in Figure 3, the symbolic semantics with assignment can avoid replicating states for each value because variables are not instantiated to values. In this section, we present a method for reducing the number of states by removing some needless internal transitions.

In order to reduce the number of states, there has been a method for finding an equal pair of states and then folding them to one state, e.g. [14]. It is, however, impossible to automatically check whether two symbolic transition graphs with assignments are equal or not in general as discussed in [9,10]. Therefore, instead of finding *all* such equal pairs, we present a method for automatically finding *some* equal pairs.

At first, we prepare some notations: the composition of locations, the independent relation between locations, the composition of symbolic internal transitions, and a consecutive symbolic transition relation, where Trns is an abbreviation of $\mathcal{E} \times \mathtt{Bexp} \times \mathtt{Assign} \times \mathtt{Loc} \times \mathcal{E}$ and it is used for expressing a subset of internal transitions.

Definition 2.1 *Let $\delta, \delta' \in \mathtt{Loc}$. The composition of locations is defined as follows:*

$$\delta\bullet\delta' = \begin{cases} (\delta_0\bullet\delta'_0\ \ \delta_1\bullet\delta'_1) & (\textit{if } \delta = (\delta_0\delta_1) \textit{ and } \delta' = (\delta'_0\delta'_1)) \\ \delta' & (\textit{if } \delta = \mathtt{0}) \\ \delta & (\textit{if } \delta' = \mathtt{0}) \\ \mathtt{1} & (\textit{otherwise}) \end{cases}$$

∎

Definition 2.2 *Let $\delta, \delta' \in \mathtt{Loc}$. The independence of locations is defined as follows:*

$$\delta \perp \delta' = \begin{cases} (\delta_0 \perp \delta'_0) \wedge (\delta_1 \perp \delta'_1) & (\textit{if } \delta = (\delta_0\delta_1) \textit{ and } \delta' = (\delta'_0\delta'_1)) \\ \mathtt{true} & (\textit{if } \delta = \mathtt{0} \textit{ or } \delta' = \mathtt{0}) \\ \mathtt{false} & (\textit{otherwise}) \end{cases}$$

∎

The composition of two locations works like the disjunction-operator, for example, $((\mathtt{01})(\mathtt{01}))\bullet((\mathtt{00})(\mathtt{11})) = ((\mathtt{01})(\mathtt{11}))$. The independence of two locations checks whether the same process participates in the two locations or not, for example,

$$((\mathtt{10})(\mathtt{01})) \perp ((\mathtt{01})\mathtt{0}) = \mathtt{true},\quad ((\mathtt{10})(\mathtt{01})) \perp ((\mathtt{01})(\mathtt{01})) = \mathtt{false}.$$

Definition 2.3 *Let $(E, b, \theta, \delta, E'), (F, b', \theta', \delta', F') \in \mathtt{Trns}$. The composition of the transitions is defined by*

$$(E, b, \theta, \delta, E') \bullet (F, b', \theta', \delta', F') = \begin{cases} (E, b \wedge b'\theta, \theta'\circ\theta, \delta\bullet\delta', F') & (\textit{if } E' \equiv F) \\ \textit{undefined} & (\textit{otherwise}) \end{cases}$$

then the composition is extended over sets of transitions $T, T' \subseteq \mathtt{Trns}$ by

$$T \bullet T' = \{tr \bullet tr' \;\mid\; tr \in T,\ tr' \in T',\ tr \bullet tr' \textit{ is defined}\}.$$

Furthermore, the iterative composition $T^\bullet$ of copies of $T \subseteq \mathtt{Trns}$ is the smallest set satisfying the following inclusions:

$$\{(E, \mathtt{true}, \varepsilon, \mathtt{0}, E) \mid E \in \mathcal{E}\} \subseteq T^\bullet \;\; \textit{and} \;\; (T^\bullet) \bullet T \subseteq T^\bullet.$$

∎

Definition 2.4 *The consecutive symbolic internal transition* $\overset{\tau[\_]/\_@\_}{\bullet\!\!\Longrightarrow} \subseteq \mathtt{Trns}$ *is defined by*

$$\overset{\tau[\_]/\_@\_}{\bullet\!\!\Longrightarrow} \;=\; (\_ \overset{\tau[\_]/\_@\_}{\bullet\!\!\twoheadrightarrow} \_)^\bullet = \{(E, b, \theta, \delta, E') \mid E \overset{\tau[b]/\theta@\delta}{\bullet\!\!\twoheadrightarrow} E'\}^\bullet.$$

Conveniently, we write $E \overset{\tau[b]/\theta@\delta}{\bullet\!\!\Longrightarrow} E'$ *for* $(E, b, \theta, \delta, E') \in \overset{\tau[\_]/\_@\_}{\bullet\!\!\Longrightarrow}$. ∎

Thus, $E \overset{\tau[b]/\theta@\delta}{\bullet\!\!\Longrightarrow} E'$ represents that if the condition b is true then E can reach to E' by zero or more internal transitions and thereafter the variables are updated by the assignment θ, where δ indicates the locations of all the processes which participate in the consecutive transition.

Next, we define a set $\mathcal{R}$ of internal transitions such that $(E, b, \theta, \delta, F) \in \mathcal{R}$ implies that E may be removed without changing behavior.

Definition 2.5 *Let* $\mathcal{R} \subseteq \overset{\tau[\_]/\_@\_}{\bullet\!\!\Longrightarrow}$. *Then $\mathcal{R}$ is a* symbolically reducible set, *if for all* $\alpha[b]/\theta@\delta$ *and E' such that* $E \overset{\alpha[b]/\theta@\delta}{\bullet\!\!\twoheadrightarrow} E'$,

> (i) *for all b_0, θ_0, δ_0, and F such that* $(E, b_0, \theta_0, \delta_0, F) \in \mathcal{R}$, *and* $\mathtt{sat}(b \wedge b_0)$,
> *if $\delta \perp \delta_0$ then for some F',* $F \overset{\alpha[b]/\theta@\delta}{\bullet\!\!\twoheadrightarrow} F'$ *and* $(E', b_0, \theta_0, \delta_0, F') \in \mathcal{R}$,
> *else* $\delta = \delta_0, \alpha = \tau$, *and* $(E, b, \theta, \delta, E') \in \mathcal{R}$,
>
> (ii) *for all b_0, θ_0, δ_0, and F' such that* $(E', b_0, \theta_0, \delta_0, F') \in \mathcal{R}$ *and* $\mathtt{sat}(b \wedge b_0)$,
> *if $\delta \perp \delta_0$ then for some F,* $F \overset{\alpha[b]/\theta@\delta}{\bullet\!\!\twoheadrightarrow} F'$ *and* $(E, b_0, \theta_0, \delta_0, F) \in \mathcal{R}$,

where $\mathtt{sat}(\_)$ *is the predicate for checking satisfiability, thus* $\mathtt{sat}(b)$ *iff* $(\exists\, \rho.\; b\rho = \mathtt{true})$. ∎

The symbolically reducible set is used for reducing needless internal transitions mainly caused by *interleaving*. For example, Figure 6(a) is the transition graph of the concurrent process $\mathtt{Abs}(x, z)$ defined in Figure 5. $\mathtt{Abs}(x, z)$ consists of two processes $\mathtt{Caller}(x)$ and $\mathtt{Callee}(z)$: $\mathtt{Callee}(z)$ returns the absolute value of x passed from $\mathtt{Caller}(x)$ if z is not zero, otherwise nondeterministically returns x or $-x$. $\mathtt{Caller}(x)$ can independently perform $\mathtt{task}$, while $\mathtt{Callee}(z)$ is checking the sign of x and reversing the sign if necessary. The independence is expressed by interleaving the events, for example the conditional internal event $\tau[\mathtt{b1}]$ can occur before and after $\mathtt{task}$ in Figure 6(a). Then, the following $\mathcal{R}$ is a symbolically reducible set:

$$\begin{aligned}\mathcal{R} = \{&(S_1, b_1, \varepsilon, (\mathtt{01}), S_3),\; (S_1, b_2, \varepsilon, (\mathtt{01}), S_4),\; (S_2, b_1, \varepsilon, (\mathtt{01}), S_5),\; (S_2, b_2, \varepsilon, (\mathtt{01}), S_6),\\ &(S_3, \mathtt{true}, (y := -y), (\mathtt{01}), S_4),\; (S_5, \mathtt{true}, (y := -y), (\mathtt{01}), S_6)\}\end{aligned}$$

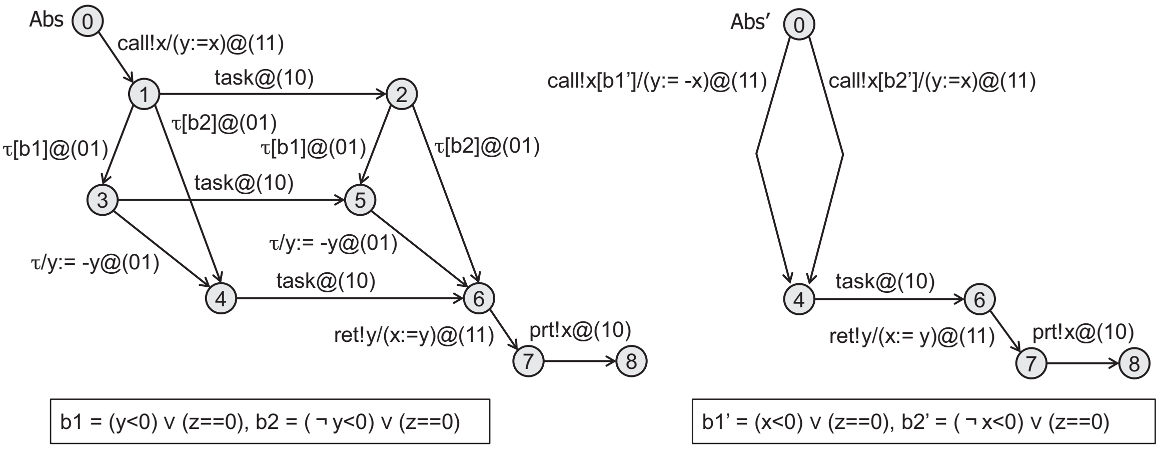
where each process (state) S_i corresponds to the node i in Figure 6(a) and S_0 is the initial state of $\mathtt{Abs}$. In this case, the states S_1 and S_3 can be bypassed[2] and removed like the transition graph shown in Figure 6(b) whose initial state is $\mathtt{Abs}'$, where Proposition 2.1 given later guarantees that $\mathtt{Abs}$ and $\mathtt{Abs}'$ are stable-failures-equivalent.

If the internal transition from S_2 to S_5 does not exist in the transition graph $\mathtt{Abs}$ of Figure 6(a), then $\mathtt{Abs}$ and $\mathtt{Abs}'$ are not stable-failures-equivalent because $\mathtt{Abs}$ can deadlock at S_2

[2] The method to bypass reducible states is given in Definition 2.6 later.

$$\begin{aligned}
\mathtt{Abs}(x,z) &= \mathtt{Caller}(x)[|\,\{|\,\mathtt{call},\mathtt{ret}\,|\}\,|](\mathtt{Callee}(z)\setminus\{|\,\mathtt{chk},\mathtt{minus}\,|\})\\
\mathtt{Caller}(x) &= \mathtt{call}!x \rightarrow \mathtt{task} \rightarrow \mathtt{ret}?x \rightarrow \mathtt{prt}!x \rightarrow \mathtt{STOP}\\
\mathtt{Callee}(z) &= \mathtt{call}?y \rightarrow \mathtt{Check}(y,z)\\
\mathtt{Check}(y,z) &= ((y{<}0 \vee z{=}{=}0)\,\&\,\mathtt{chk} \rightarrow \mathtt{minus} \rightarrow \mathtt{Ret}(-y)) \;\Box\; ((\neg(y{<}0) \vee z{=}{=}0)\,\&\,\mathtt{chk} \rightarrow \mathtt{Ret}(y))\\
\mathtt{Ret}(y) &= \mathtt{ret}!y \rightarrow \mathtt{STOP}
\end{aligned}$$

Figure 5. A concurrent process `Abs`(x,z)



(a) the original transition graph of Abs

(b) the reduced transition graph

Figure 6. The basic idea for reducing the number of transitions

if $x = 0$ and $z \neq 0$. The condition (i) in Definition 2.5 requires that the internal transition from S_2 to S_5 must exist if the internal transition from S_1 to S_3.

On the other hand, if the internal transition from S_1 to S_3 does not exist, the state S_1 must remain for the case $\neg b_2$ even after making a bypass from S_0 to S_4. It means that the nondeterminism in the case $z = 0$ disappears by the bypass (i.e. if $z = 0$ then x is deterministically returned), thus `Abs` and `Abs`$'$ are not stable-failures-equivalent. The condition (ii) in Definition 2.5 requires that the internal transition from S_1 to S_3 must exist if the internal transition from S_2 to S_5 exists.

In Figure 6, it is noted that `Abs` and `Abs`$'$ are *not* weakly bisimilar [3] because `Abs` has a nondeterministic choice at S_1 after `call` if $z = 0$. This is an important reason why stable-failures-equivalence is used in this paper.

Then, the process-name $\mathtt{BP}^{(n)}_{(\mathcal{R},E)}(\tilde{x})$ is defined for *bypassing* reducible states.

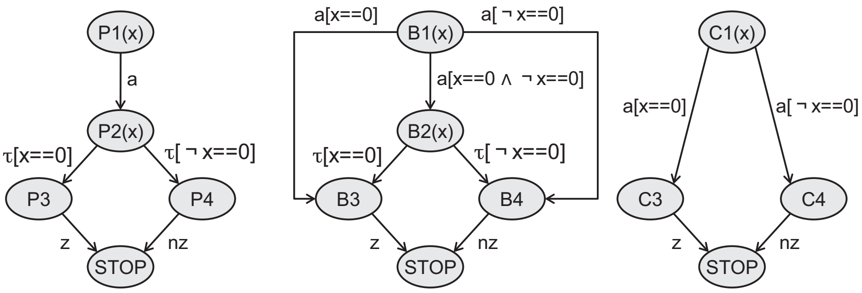
Definition 2.6 *Let $\mathcal{R}$ be a symbolically reducible set and $n \in \mathtt{Nat}$. Then, the* bypassed transition relation $\bullet\!\!\longrightarrow\!\!\!\!\rightarrow_{(\mathcal{R},n)} \subseteq \mathcal{E} \times \mathtt{Act} \times \mathcal{E}$ *with respect to $\mathcal{R}$ and n is the smallest relation satisfying the following rules:*

- (`base`)*: if* $E \bullet\!\!\xrightarrow{\alpha[b]/\theta@\delta}\!\!\!\!\rightarrow E'$ *then* $E \bullet\!\!\xrightarrow{\alpha[b]/\theta@\delta}\!\!\!\!\rightarrow_{(\mathcal{R},0)} E'$
- (`bypass`)*: if* $(E \bullet\!\!\xrightarrow{\alpha[b]/\theta@\delta}\!\!\!\!\rightarrow_{(\mathcal{R},n)} E' \;\wedge\; (E', b_0, \theta_0, \delta_0, E'') \in \mathcal{R} \;\wedge\; \mathtt{bv}(\alpha) \cap \mathtt{fv}(b_0\theta) = \emptyset)$

  then $E \bullet\!\!\xrightarrow{\alpha[b \wedge b_0\theta]/\theta_0\circ\theta@(\delta\bullet\delta_0)}\!\!\!\!\rightarrow_{(\mathcal{R},n+1)} E''$*,*
- (`rest`)*: if* $E \bullet\!\!\xrightarrow{\alpha[b]/\theta@\delta}\!\!\!\!\rightarrow_{(\mathcal{R},n)} E'$ *then* $E \bullet\!\!\xrightarrow{\alpha[b \wedge \mathtt{rest}_{(\mathcal{R},E,\alpha,\theta)}]/\theta@\delta}\!\!\!\!\rightarrow_{(\mathcal{R},n+1)} E'$*,*

where $\mathtt{rest}_{(\_)}$ *is the Boolean expression defined by*

$$\mathtt{rest}_{(\mathcal{R},E,\alpha,\theta)} = \bigwedge\{\neg b_0 \mid \exists\, \theta_0, \delta_0, E'.\ (E, b_0, \theta_0, \delta_0, E') \in \mathcal{R},\ \mathtt{bv}(\alpha) \cap \mathtt{fv}(b_0\theta) = \emptyset\}.$$



(a) original graph (b) bypassed graph (c) cleaned graph

Figure 7. The bypassed transition graph (every location is 1 and is omitted)

Then, by using $\bullet\!\!\twoheadrightarrow_{(\mathcal{R},n)}$*, the* bypassed process $\mathtt{BP}^{(n)}_{(\mathcal{R},E)}(\tilde{x})$ *is defined as follows:*

$$\mathtt{BP}^{(n)}_{(\mathcal{R},E)}(\tilde{x}) = \mathtt{SP}_{(\bullet\!\!\twoheadrightarrow_{(\mathcal{R},n)},E)}(\tilde{x})$$

where $\mathtt{SP}_{(\ldots,E)}(\tilde{x})$ *is the process-name defined in Definition 1.4.* ∎

The first rule (base) means that $\bullet\!\!\twoheadrightarrow_{(\mathcal{R},0)}$ is exactly $\bullet\!\!\twoheadrightarrow$. The second rule (bypass) is used for bypassing E' by the action $\alpha[b \wedge b_0\theta]/\theta_0\circ\theta@(\delta\bullet\delta_0)$ if $E \overset{\alpha[b]/\theta@\delta}{\bullet\!\!\twoheadrightarrow}_{(\mathcal{R},n)} E'$ and $(E', b_0, \theta_0, \delta_0, E'') \in \mathcal{R}$, and the value received by α does not affect the next condition b_0 (i.e. $\mathtt{bv}(\alpha) \cap \mathtt{fv}(b_0\theta) = \emptyset$). The third rule (rest) is used for strengthening the condition of the existing original transition by $\mathtt{rest}_{(\_)}$. It means that the original transition can be performed only if the condition of every bypassed-transition is false (i.e. $\mathtt{rest}_{(\_)} = \mathtt{true}$). For example, in the transition graph of Figure 7(a), the following set $\mathcal{R}$ is a symbolically reducible set.

$$\mathcal{R} = \{(\mathtt{P2}, x = 0, \varepsilon, \mathtt{1}, \mathtt{P3}),\ (\mathtt{P2}, x \neq 0, \varepsilon, \mathtt{1}, \mathtt{P4})\}$$

Then, Figure 7(b) shows that the process B1 generated from P1 by bypassing the reducible state P2 (i.e. $\mathtt{B1} = \mathtt{BP}^{(1)}_{(\mathcal{R},\mathtt{P1})}(x)$). Here, it is important to note that the transition from B1 to B2 is never performed. Therefore, it can be removed as shown Figure 7(c). This transformation seems to be easy, but it is difficult to find such reducible states and to bypass transitions because there are generally many interleaving transitions in concurrent processes.

Then, in order to prove that the original process and the bypassed process are *stable-failures-equivalent* [2], we give the following lemma.

Lemma 2.1 *Let* $t \in \mathtt{Act}_0^*$ *and* $\mathcal{R}$ *be a symbolically reducible set. If* $E\rho \xrightarrow{t} P'$*, then for some* E' *and* ρ'*,* $P' \equiv E'\rho'$ *and for all* b_0*,* θ_0*,* δ_0*, and* F' *such that* $(E', b_0, \theta_0, \delta_0, F') \in \mathcal{R}^\bullet$ *and* $b_0\rho'$*, for some* b'_0*,* θ'_0*,* δ'_0*, and* F''*,* $(E', b'_0, \theta'_0, \delta'_0, F'') \in \mathcal{R}^\bullet$*,* $b'_0\rho'$*,* $(E', b_0, \theta_0, \delta_0, F') \asymp (E', b'_0, \theta'_0, \delta'_0, F'')$*, and* $(\mathtt{BP}^{(n)}_{(\mathcal{R},E)}(\tilde{x})\backslash\{\mathtt{tmp}\})\rho \overset{\widehat{t}}{\Longrightarrow} (\mathtt{BP}^{(n)}_{(\mathcal{R},F'')}(\tilde{x}')\backslash\{\mathtt{tmp}\})(\theta'_0\circ\rho')$*, where* $tr_1 \asymp tr_2$ *represents that* tr_1 *and* tr_2 *are comparable (i.e.* $tr_1 \preceq tr_2 \vee tr_2 \preceq tr_1$*) by the partial order* $\preceq$ *defined by:* $tr_1 \preceq tr_2 \Leftrightarrow \exists\, tr.\ tr_1 \bullet tr = tr_2$*.* [3]

Proof: This lemma can be proved by induction on the length of t and n. Especially, the following sublemma is the key for proving it at the last transition of t.

Sublemma: If $E\rho \xrightarrow{\alpha} P'$ then for some E' and ρ', $P' \equiv E'\rho'$ and
for all b'_0, θ'_0, δ'_0, and F'_0 such that $(E', b'_0, \theta'_0, \delta'_0, F'_0) \in \mathcal{R}^\bullet$ and $b'_0\rho'$,
for some b_0, θ_0, δ_0, and F_0, $(E, b_0, \theta_0, \delta_0, F_0) \in \mathcal{R}^\bullet$, $b_0\rho$ and
for all b_1, θ_1, δ_1, and F such that $(E, b_0, \theta_0, \delta_0, F_0) \asymp (E, b_1, \theta_1, \delta_1, F) \in \mathcal{R}^\bullet$ and $b_1\rho$,

[3] $P \xrightarrow{t} P'$ is the sequential standard transition from P to P' by $t \in \mathtt{Act}_0^*$, $\widehat{t}$ is the event-sequence obtained from t by deleting τ, and $P \overset{t}{\Longrightarrow} P'$ is the weak standard transition, where zero or more internal transitions can be inserted between observable transitions (e.g. see [3] for the weak standard transition).

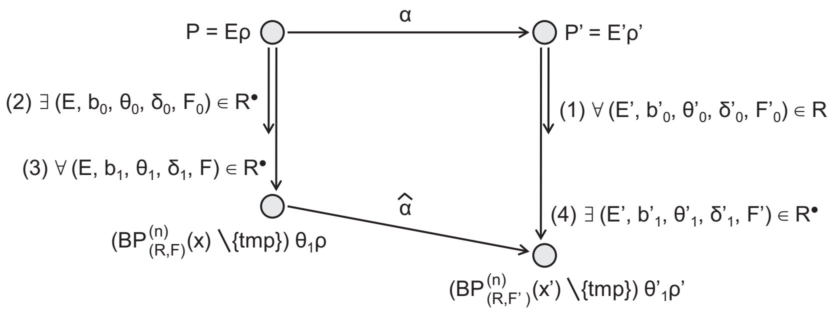

$P = E\rho$
α
$P' = E'\rho'$
(2) $\exists\, (E, b_0, \theta_0, \delta_0, F_0) \in R^\bullet$
(1) $\forall\, (E', b'_0, \theta'_0, \delta'_0, F'_0) \in R^\bullet$
(3) $\forall\, (E, b_1, \theta_1, \delta_1, F) \in R^\bullet$
$\widehat{\alpha}$
(4) $\exists\, (E', b'_1, \theta'_1, \delta'_1, F') \in R^\bullet$
$(BP^{(n)}_{(R,F)}(x) \setminus\{tmp\})\, \theta_1\rho$
$(BP^{(n)}_{(R,F')}(x') \setminus\{tmp\})\, \theta'_1\rho'$

Figure 8. The relation between E and $(\mathtt{BP}^{(n)}_{(\mathcal{R},F)}(\tilde{x}))\backslash\{\mathtt{tmp}\}$ in the sublemma in Lemma 2.1

for some b'_1, θ'_1, δ'_1, and F', $(E', b'_0, \theta'_0, \delta'_0, F'_0) \asymp (E', b'_1, \theta'_1, \delta'_1, F')$, $b'_1\rho'$, and $(\mathtt{BP}^{(n)}_{(\mathcal{R},F)}(\tilde{x})\backslash\{\mathtt{tmp}\})\theta_1\rho \overset{\widehat{\alpha}}{\Longrightarrow} (\mathtt{BP}^{(n)}_{(\mathcal{R},F')}(\tilde{x}')\backslash\{\mathtt{tmp}\})\theta'_1\rho'$.

When comparing the event-sequences of original process E and the bypassed process $(\mathtt{BP}^{(n)}_{(\mathcal{R},F)}(\tilde{x}))\backslash\{\mathtt{tmp}\}$, it is noted that E has to perform more internal events than the bypassed process because the bypassed process can bypass reducible states. Therefore, $\widehat{\alpha}$ is used in the bypassed process for deleting the extra internal event τ of E (note: $\widehat{\tau} = \varepsilon$). The relation between E and $(\mathtt{BP}^{(n)}_{(\mathcal{R},F)}(\tilde{x}))\backslash\{\mathtt{tmp}\}$ is shown in Figure 8. See the proof-note in the CONPASU website [15] for the details. ∎

A bypass often makes the other bypasses possible. Therefore, the bypassed process $\mathtt{BP}^{(n)}_{(\mathcal{R},E)}(\tilde{x})$ has the parameter n for iteratively bypassing by using the set $\mathcal{R}$. It is important that $\mathcal{R}$ can be *reused* at each step n in the iterative bypass, in other words, it is not necessary to compute $\mathcal{R}$ for each step.

Then, we present Proposition 2.1 which guarantees that the original process and the bypassed process are *stable-failures-equivalent* [2].

Proposition 2.1 *Let* $E \in \mathcal{E}$, $n \in \mathtt{Nat}$, *and* $\mathcal{R}$ *be a symbolically reducible set. Then,*

$$E =_{\mathcal{F}} \mathtt{BP}^{(n)}_{(\mathcal{R},E)}(\tilde{x})\backslash\{\mathtt{tmp}\}$$

Proof: Lemma 2.1 implies $\mathtt{traces}(E\rho) \subseteq \mathtt{traces}((\mathtt{BP}^{(n)}_{(\mathcal{R},E)}(\tilde{x})\backslash\{\mathtt{tmp}\})\rho)$, where $\mathtt{traces}(P)$ is the set of traces of P. The opposite direction "$\supseteq$" can be proven by the following sublemma:

> **Sublemma**: if $(\mathtt{BP}^{(n)}_{(\mathcal{R},E)}(\tilde{x})\backslash\{\mathtt{tmp}\})\rho \xrightarrow{\alpha} P'$, then for some E' and ρ', $E\rho \xrightarrow{\alpha}\overset{\varepsilon}{\Longrightarrow} E'\rho'$ and $P' \equiv (\mathtt{BP}^{(n)}_{(\mathcal{R},E')}(\tilde{x}')\backslash\{\mathtt{tmp}\})\rho'$.

This sublemma is easier than Lemma 2.1. Furthermore, it is easy to show that E' and $\mathtt{BP}^{(n)}_{(\mathcal{R},E')}(\tilde{x})\backslash\{\mathtt{tmp}\}$ have the same refusals for any E'. Here, F'' of $\mathtt{BP}^{(n)}_{(\mathcal{R},F'')}(\tilde{x}')\backslash\{\mathtt{tmp}\}$ in Lemma 2.1 is not necessarily the same as E', but if E' is stable (i.e. has no internal transition), then $E' \equiv F''$ because $(E', b'_0, \theta'_0, \delta'_0, F'') \in \mathcal{R}^\bullet \subseteq \overset{\tau[\_]/\_@\_}{\bullet\!\!\Longrightarrow}$. Hence, $\mathtt{failures}(E\rho) = \mathtt{failures}((\mathtt{BP}^{(n)}_{(\mathcal{R},E)}(\tilde{x})\backslash\{\mathtt{tmp}\})\rho)$, where $\mathtt{failures}(P)$ is the set of failures of P. ∎

By Proposition 2.1, internal transitions in $\mathcal{R}$ can be bypassed with preserving the behavior up to stable-failures-equivalence. In order to apply the proposition, however, it is necessary to find a symbolically reducible set $\mathcal{R}$, according to Definition 2.5. In general, it is difficult to find the largest reducible set because $\overset{\tau[\_]/\_@\_}{\bullet\!\!\Longrightarrow}$ may be infinite by compositions of assignments (e.g. the infinite composition of $(n := n + 1)$) even if the symbolical transition graph is finite. Therefore, we present a method for generating a reducible set, which is not necessarily largest.

Definition 2.7 *Let $\mathcal{S} \subseteq \mathcal{E}$. Then, the set $\mathtt{SR}_{\mathcal{S}} \subseteq \overset{\tau[\_]/\_@\_}{\bullet\!\!\Longrightarrow\!\!\!\!\Rightarrow}$ is defined as follows:*

$$
\begin{aligned}
\mathtt{SR}_{\mathcal{S}} &= \textstyle\bigcap_{n\geq 0} \mathtt{SR}_{\mathcal{S}}^{(n)}\ ,\\
\mathtt{SR}_{\mathcal{S}}^{(0)} &= \{(E, b_0, \theta_0, \delta_0, F) \mid E \in \mathcal{S},\ E \overset{\tau[b_0]/\theta_0@\delta_0}{\bullet\!\!\longrightarrow\!\!\!\!\rightarrow} F,\ E \not\equiv F\},\\
\mathtt{SR}_{\mathcal{S}}^{(n+1)} &= \mathtt{SRB}(\mathtt{SRF}(\mathtt{SR}_{\mathcal{S}}^{(n)})),
\end{aligned}
$$

where $\mathtt{SRF}(\mathcal{R}), \mathtt{SRF}(\mathcal{R}) \subseteq \overset{\tau[\_]/\_@\_}{\bullet\!\!\Longrightarrow\!\!\!\!\Rightarrow}$ are defined as follows:

$$
\begin{aligned}
\mathtt{SRF}(\mathcal{R}) = \{(E, b_0, \theta_0, \delta_0, F) \in \mathcal{R} \mid{}& \forall\, \alpha, b, \theta, \delta, E'.\ (\mathtt{wsat}(b \wedge b_0),\ E \overset{\alpha[b]/\theta@\delta}{\bullet\!\!\longrightarrow\!\!\!\!\rightarrow} E')\\
&\Rightarrow (\textit{if}\ \delta \perp \delta_0\ \textit{then}\ (\exists\, F'.\, F \overset{\alpha[b]/\theta@\delta}{\bullet\!\!\longrightarrow\!\!\!\!\rightarrow} F',\ (E', b_0, \theta_0, \delta_0, F') \in \mathcal{R})\\
&\qquad \textit{else}\ (\delta = \delta_0,\ \alpha = \tau,\ (E, b, \theta, \delta, E') \in \mathcal{R}))\}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{SRB}(\mathcal{R}) = \{(E', b_0, \theta_0, \delta_0, F') \in \mathcal{R} \mid{}& \forall\, \alpha, b, \theta, \delta, E.\ (\mathtt{wsat}(b \wedge b_0),\ E \overset{\alpha[b]/\theta@\delta}{\bullet\!\!\longrightarrow\!\!\!\!\rightarrow} E')\\
&\Rightarrow (\textit{if}\ \delta \perp \delta_0\ \textit{then}\ (\exists\, F.\, F \overset{\alpha[b]/\theta@\delta}{\bullet\!\!\longrightarrow\!\!\!\!\rightarrow} F',\ (E, b_0, \theta_0, \delta_0, F) \in \mathcal{R}))\}
\end{aligned}
$$

where $\mathtt{wsat}(\_)$ is a predicate such that if b is satisfiable then $\mathtt{wsat}(b)$ is true. ▮

The set $\mathcal{S}$ in Definition 2.7 is usually the set of all the reachable states from the initial process. The sets $\mathtt{SRF}(\mathcal{R})$ and $\mathtt{SRB}(\mathcal{R})$ are used for removing internal transitions which do not satisfy the conditions (i) and (ii) in Definition 2.5, respectively. Here, it is noted that if it is hard to decide the satisfiability of b then $\mathtt{wsat}(b)$ can be $\mathtt{true}$ for the safety because only one direction (b is satisfiable $\Rightarrow \mathtt{wsat}(b)$) is required in Definition 2.7. It is useful for implementing an automatic tool based on Definition 2.7.

By Definition 2.7, since $\mathtt{SRF}(\mathcal{R}) \subseteq \mathcal{R}$ and $\mathtt{SRB}(\mathcal{R}) \subseteq \mathcal{R}$ (i.e. $\mathtt{SR}_{\mathcal{S}}^{(n+1)} \subseteq \mathtt{SR}_{\mathcal{S}}^{(n)}$) for any n, there is necessarily a natural number m such that $\mathtt{SR}_{\mathcal{S}}^{(m)} = \mathtt{SR}_{\mathcal{S}}^{(m+1)}$ if the set $\mathcal{S}$ of reachable states is finite. Then, the expected proposition is presented.

Proposition 2.2 *If $\mathtt{SR}_{\mathcal{S}}^{(m)} = \mathtt{SR}_{\mathcal{S}}^{(m+1)}$, then $\mathtt{SR}_{\mathcal{S}}^{(m)}$ is a symbolically reducible set.*
Proof: It can be shown that the following $\mathcal{R}$ is a symbolically reducible set.

$$\mathcal{R} = \{(E, b, \theta, \delta.F) \mid \exists\, m.\, (E, b, \theta, \delta, F) \in \mathtt{SR}_{\mathcal{S}}^{(m)} = \mathtt{SR}_{\mathcal{S}}^{(m+1)}\}$$

It is not difficult because the conditions in Definition 2.7 imply ones in Definition 2.5. ▮

Consequently, by Propositions 2.1 and 2.2, the following corollary is derived.

Corollary 2.1 *Assume that $n, m \in \mathtt{Nat}$, $E \in \mathcal{E}$, $\mathcal{R} = \mathtt{SR}_{\mathcal{S}}^{(m)} = \mathtt{SR}_{\mathcal{S}}^{(m+1)}$, and $\mathcal{S} \subseteq \mathcal{E}$ is the set of reachable states from E. Then, $E =_{\mathcal{F}} \mathtt{BP}_{(\mathcal{R},E)}^{(n)}(\tilde{x}) \backslash \{\mathtt{tmp}\}$.*

The set $\mathtt{SR}_{\mathcal{S}}$ is not necessarily the largest reducible set. Interactive theorem provers like Isabelle [16] may allow us to find such largest reducible sets. In this paper, however, we are more interested in *automatically reducing* the number of states than *semi-automatically minimizing* it. Therefore, our reduction method is sound but not complete. We, however, expect that the method can remove many needless internal transitions caused by interleaving. It is demonstrated in Sections 3 and 4 by implementing the method.

3. CONPASU-tool: an Implementation

We have implemented the analysis-method, which is presented in Sections 1 and 2, in a prototype-tool called CONPASU (CONcurrent Process Analysis SUpport tool) in Java (cur-

```
  CAL(N) = (SQREM(N) [|{|rem,end2|}|] SUM(0)) \ {|rem,end2|}
SQREM(n) = (SQ(n) [|{|sq,end1|}|] REM) \ {|sq,end1|}
   SQ(n) = ((n>0) & in?x1 -> sq!(x1*x1) -> SQ(n-1)) [] ((n==0) & end1!0 -> STOP)
     REM = sq?x2 -> rem!(x2%10) -> REM [] end1?z1 -> end2!z1 -> STOP
  SUM(y) = rem?x3 -> prt!x3 -> SUM(y+x3) [] end2?z2 -> prts!y -> STOP
```

Figure 9. The concurrent process CAL(N) (a readable script by CONPASU)

rently about 6,000 lines). It means that CONPASU is a tool for generating a sequential process E from each concurrent process F such that $E =_{\mathcal{F}} F$. It can also generate a script in the DOT language [17] for drawing the symbolic transition graph of the generated sequential process, for example by using Graphviz (Graph Visualization Software) [18].

In this section, it is explained by the example CAL(N) in Figure 9 how to use CONPASU for analyzing concurrent processes , where N is the initial value of n in SQ(n). The input-language of CONPASU is a sub-language of CSP$_M$ (Machine-readable dialect of CSP) used in FDR [4], and CONPASU can directly read the script of Figure 9. The concurrent process CAL(N) consists of three processes: SQ(n), REM, and SUM(y). The process SQ(n) has been explained in Section 1. The processes REM and SUM(y) behave as follows. If REM receives a value from the channel sq, then sends the remainder of dividing the value by 10 and then returns to REM, and if it receives a value from the channel end1, then forwards it to the channel end2 and then stops. If SUM(y) receives a value, to which x is bound, from the channel rem, then prints it and behaves like SUM(y + x), and if it receives a value from the channel end2, then prints y and then stops

At first, Figure 10 shows the transition graph generated from CAL(N) by CONPASU, according to the symbolic operational semantics with assignments and locations in Definition 1.3, where Graphviz [18] is used for drawing the graph. And Figure 11 shows the reduced transition graph generated from the graph in Figure 10 by CONPASU, according to the state-reduction method presented in Section 2. By the state-reduction, the numbers of states and transitions decrease by 5 (from 12 to 7) and 7 (from 17 to 10), respectively.

Figure 11 is useful for understanding the whole behavior of the concurrent process CAL(N). However, it is not avoidable that transition graphs become very complex for large scale systems even if the state-reduction is applied. In such cases, more abstract behaviors can be extracted by hiding uninteresting events. For example, although CAL(N) prints a value by prt at each receiving, we can see the abstract behavior by focusing on the input in and the final result prts, in other words, by hiding prt as follows.

```
ACAL(N) = CAL(N)\{|prt|}
```

In this case, ACAL(N) is expected to behave like the specification SPEC(N) in Figure 12. The specification means that SPEC(N) iteratively receives a value, to which x is bound, and adds x*x%10 to the variable y, n-times, and thereafter y is printed by prts. In fact, the model checker FDR can verify that ACAL(N) and SPEC(N) are stable-failures-equivalent by fixing the initial value N and finitising the range of the input and the variable y. The specification SPEC(N) is simple and is easily described. It is, however, sometimes more difficult to describe such specifications than implementations.

CONPASU can automatically generate specifications (in CSP$_M$ script) from implementations. Figures 13 and 14 show the transition graph of ACAL(N) after state-reduction and the sequential process S(N), respectively, generated by CONPASU. Therefore, it is guaranteed by Corollary 2.1 that S(N) and ACAL(N) are stable-failures-equivalent for any initial value N and any input. By comparing the generated specification S(N) with the ideal specification SPEC(N), S(N) has an extra internal transition (from S0(n, y) to S4(n, y)), but it is easy to man-

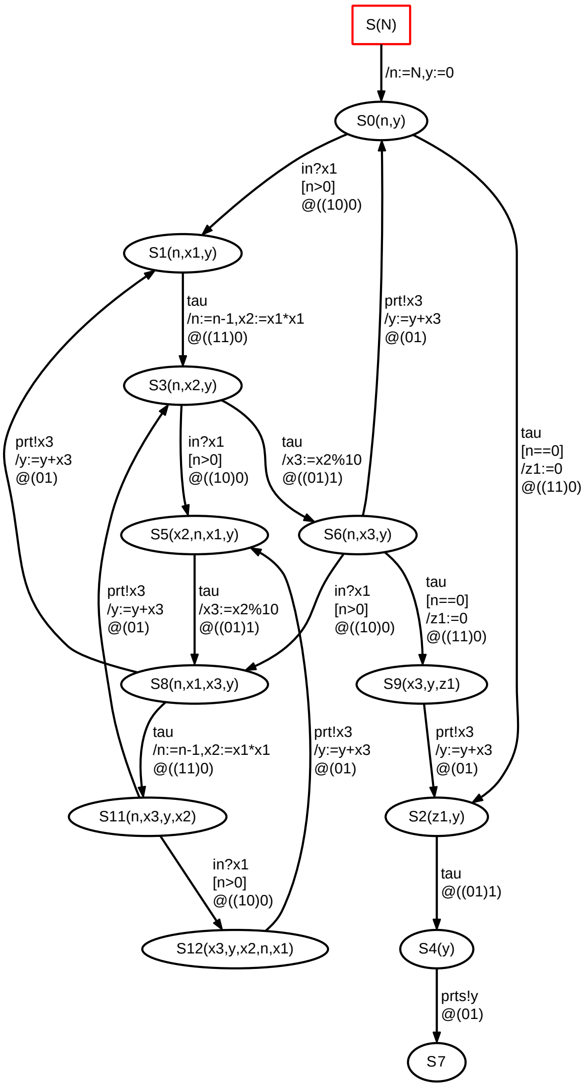


Figure 10. The symbolic transition graph of CAL(N) (the top box points to the initial state)

ually prove that S(N) and SPEC(N) are stable-failures-equivalent for any initial value N and any input. As shown in this example, specifications generated by CONPASU are not necessarily ideal. However, such generated specifications are helpful for formally describing ideal specifications used in FDR. CONPASU will be also used as a support tool of FDR when formally describing specifications of concurrent processes. We are now considering how to improve the analysis-method of CONPASU for generating more ideal specifications.

CONPASU is still a prototype and has not been polished yet. The current CONPASU soundly checks the *unsatisfiability* of Boolean expressions by transforming them to disjunctive normal forms, and it is not complete. The incompleteness, however, does not invalidate Corollary 2.1 because Definition 2.7 only requires that if $\neg\mathtt{wsat}(b)$ is true then b is unsatisfiable. Furthermore, the syntactical identity is used for the equality over data-expressions (e.g. $x + 1 \neq 1 + x$). The syntactical identity seems strong, but it is expected to be still useful for reducing many transitions caused by interleaving, and it has been shown in the example CAL(N) and it is also demonstrated in Section 4.

4. Application

In this section, we demonstrate how CONPASU analyzes concurrent processes by using the example TransferSys given in Figure 15. It is a system for transferring data-sequences

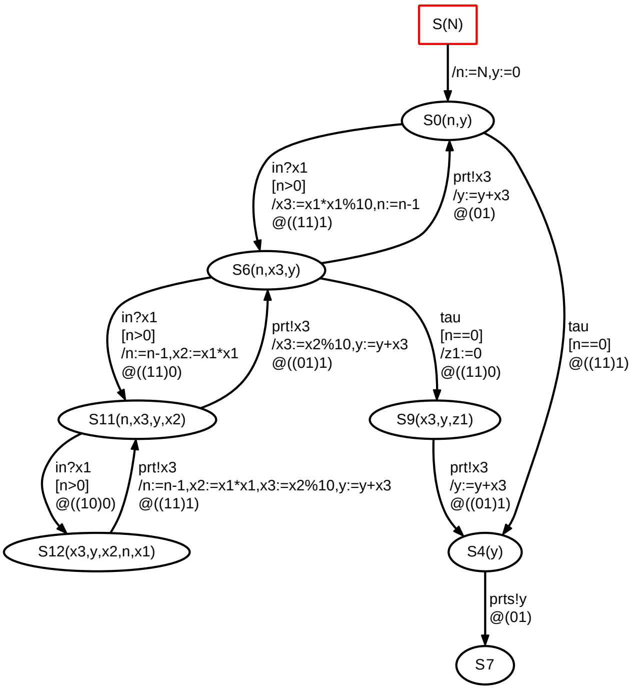


Figure 11. The symbolic transition graph of `CAL(N)` after state-reduction

```
  SPEC(N) = LOOP(N,0)
LOOP(n,y) = (n>0) & in?x -> LOOP(n-1,y+x*x%10) [] (n==0) & prts!y -> STOP
```

Figure 12. The expected specification of the abstract concurrent process `ACAL(N)`

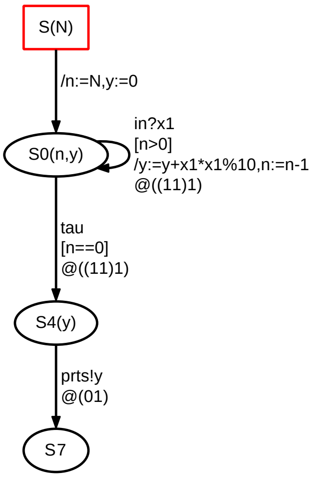


Figure 13. The symbolic transition graph of `ACAL(N)`

```
S(N)  = S0(N,0)\{|tmp|}
S0(n,y)  = (n>0) & in?x1 -> S0(n-1,y+x1*x1%10) [] (n==0) & tmp!0 -> S4(y)
S4(y)  = prts!(y) -> S7
S7  = STOP
```

Figure 14. A specification `S(N)` generated from `ACAL(N)` by `CONPASU`

```
TransferSys = (UI [|{|input,quit0,succ,ok,ng|}|] Transfer)
                  \ {|input,quit0,succ,ok,ng|}
   Transfer = (Sender [|{|start,net,term,quit1,ack|}|] Receiver)
                      \ {|start,net,term,quit1,ack|}

             UI = upload?ds -> input!ds -> (ok?a -> Wait [] ng?a -> UI)
           Wait = (cancel?b -> quit0!0 -> UI)
                []((succ?u -> complete!0 -> UI)

         Sender = input?ds0 -> Check(ds0)
     Check(ds0) = ((#ds0>0) & ok!0 -> start!0 -> Sending(ds0))
                []((not #ds0>0) & ng!0 -> Sender)
   Sending(ds0) = ((#ds0>0) & net!(head(ds0)) -> Sending(tail(ds0)))
                []((not #ds0>0) & term!0 -> Term)
                [](quit0?x -> quit1!0 -> Sender)
           Term = ack?z -> (succ!0 -> Sender [] quit0?x -> Sender)

       Receiver = start?y -> Receiving(<>)
 Receiving(ds1) = (net?d -> Receiving(ds1^<d>))
                [](term?y -> output!ds1 -> ack!0 -> Receiver)
                [](quit1?y -> Receiver)
```

Figure 15. The CSP$_M$-script of the system `TransferSys` for transferring data-sequences with UI

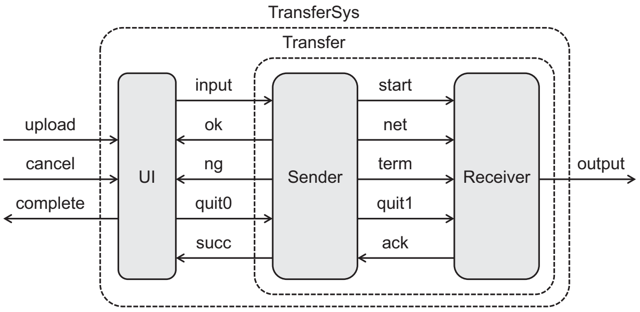


Figure 16. The structure of the transfer system `TransferSys`

from the process `Sender` to the process `Receiver`. The process `UI` is the user-interface for controlling `Sender`. The structure of `TransferSys` is shown in Figure 16.

The system `TransferSys` behaves as follows:

- **Start phase**: process `UI` receives a data-sequence from `upload` and then sends it to the channel `input`. The process `Sender` checks the length of the data-sequence received from `input`, and if the length is greater than zero then `Sender` replies `ok` to `UI` and thereafter sends the start signal `start` and moves to the transfer phase, otherwise `Sender` replies `ng` to `UI` and thereafter both processes returns to the initial states. If `Receiver` receives the start signal, then it initialises the data-sequence `ds1` to the empty sequence <>, and moves to the transfer phase.
- **Transfer phase**: after starting the transfer, if the length of the data-sequence is greater than zero (`#ds0>0`), then `Sender` iteratively sends the first data `head`(`ds0`) to the channel `net` and retains the remain `tail`(`ds0`), otherwise sends the terminal signal `term` and moves to the termination phase. At the same time, if `Receiver` receives a data from `net`, then adds it to the sequence by `ds1^<d>`, and if `Receiver` receives the terminal signal then moves to the termination phase.
- **Termination phase**: process `Receiver` sends the data-sequence `ds1` to the channel `output` and then acknowledges completion to `Sender`. After receiving the acknowl-

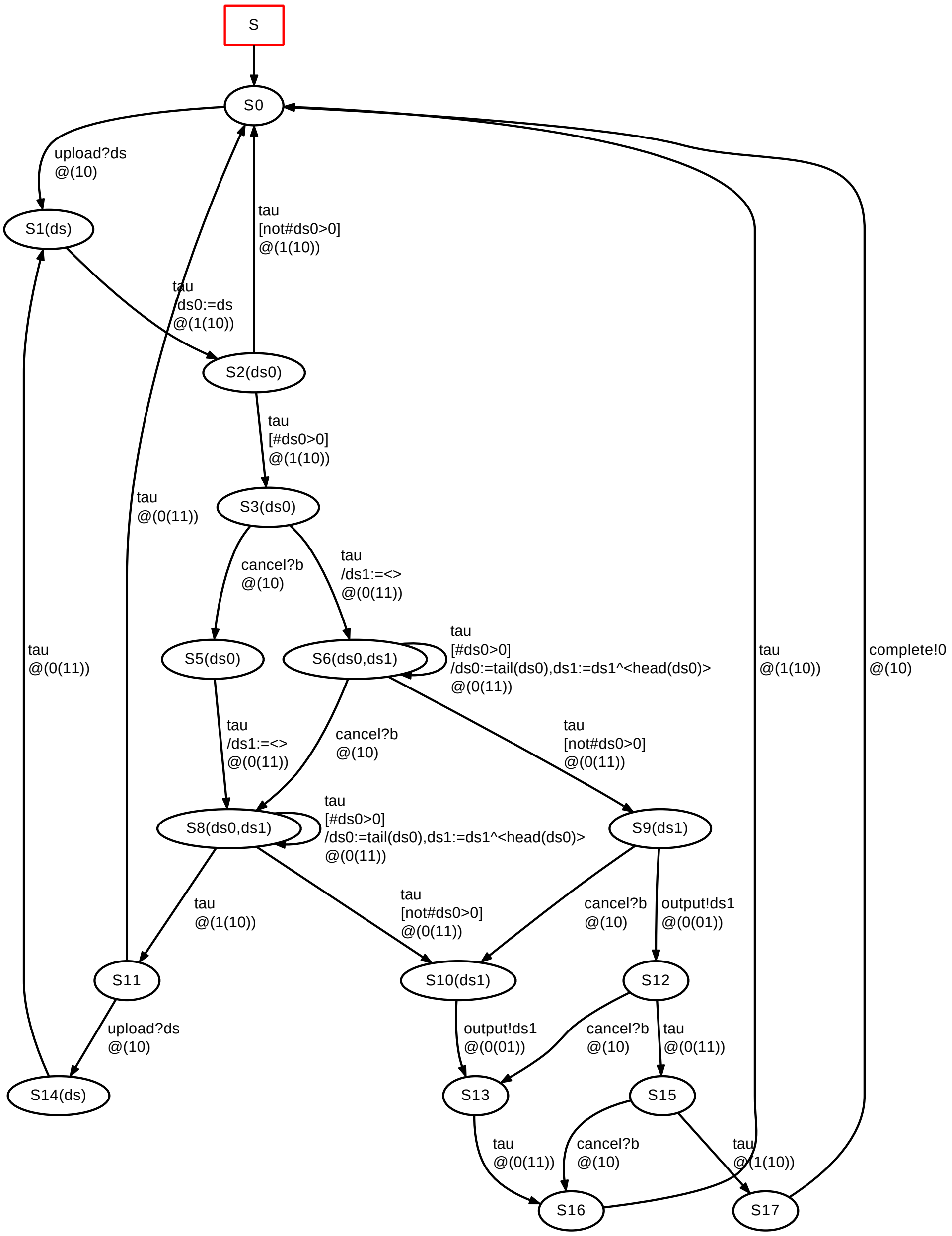


Figure 17. The symbolic transition graph of `TransferSys` in Figure 15

edgment, `Sender` replies with success to `UI`, and then `UI` reports success to users.

- **Cancel**: users can cancel the transfer by the channel `cancel` in the transfer phase. The cancel is forwarded to `Sender` and `Receiver` by the quit-signals `quit0` and `quit1`.

Since the processes have a lot of interactions in the system `TransferSys`, it is not easy to understand the whole behavior. Figure 17 shows the transition graph derived from the CSP$_M$-script in Figure 15 by the symbolic operational semantics in Definition 1.3. Then, it can be reduced to the graph in Figure 18 by hiding the channel `complete` and reducing internal transitions by Corollary 2.1. These graphs can be automatically generated by `CONPASU`. In Figure 18, we can know how `TransferSys` behaves, for example, as follows:

- The label on the loop from/to the state `S6`,

  ```
  tau[#ds0>0]/ds0:=tail(ds0),ds1:=ds1^<head(ds0)>@(0(11))
  ```

  means that if the length of the data-sequence `ds0` held in `Sender` is greater than `0`, then the first data is attached to the tail of `ds1` held in `Receiver`.

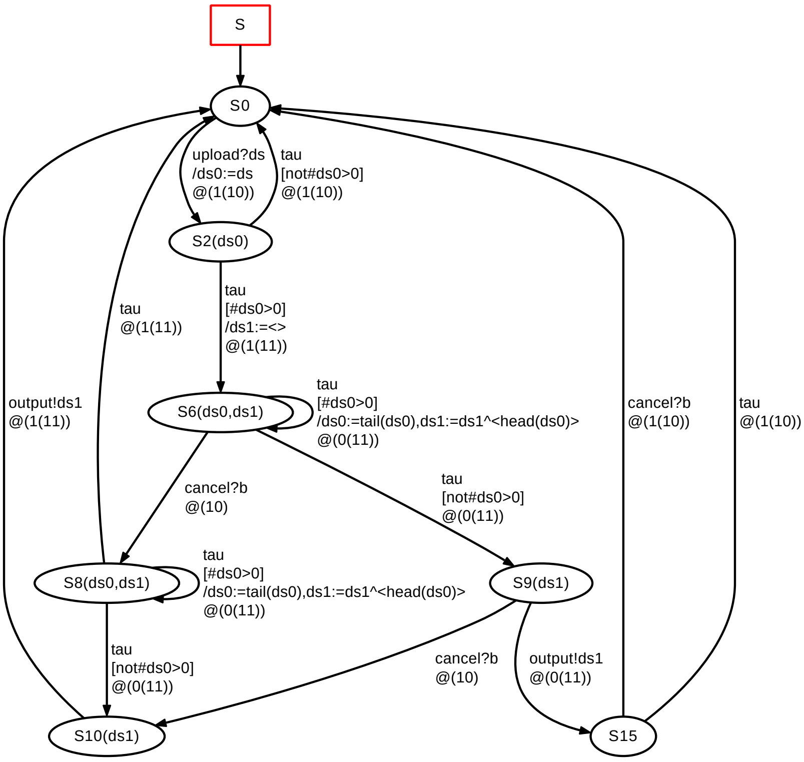


Figure 18. The reduced symbolic transition graph of `TransferSys` $\setminus$ {| `complete` |}

- The loop from/to the state `S8` means that data may be transferred even after the cancel because forwarding the quit-signals may delay.

In the end of this section, the termination phase of `Sender` is reconsidered. The process `Term` in Figure 15 can receive the quit signal `quit0` even after receiving the acknowledgment. It seems needless, but the system `TransferSys'`, which is the same as `TransferSys` except that `Term` is replaced by the following `Term'`

```
Term' = ack?z -> succ!0 -> Sender,
```

has a deadlock because it is possible to perform the cancel just after the successful termination. Figure 19 is the reduced transition graph generated from `TransferSys'\{|complete|}`, and it shows how the system reaches to the deadlock state `S16`.

5. Related Work

There are various model checkers for process algebra, for example, FDR [4], PAT [5], CWB [6], and mCRL2 [7]. The main purpose of such model checkers is to check equalities or refinements between an implementation and a specification. On the other hand, the main purpose of this work is to automatically generate a specification (an abstract sequential process) from an implementation (a concurrent process).

Some model checkers provide functionality to display transition graphs. For example, Figures 20 and 21 show two transition graphs of `CAL(N)`, introduced in Section 3 (see Figure 9), displayed by PAT [5] and LTSA [19], respectively. In Figures 20 and 21, the number of states are 105 in PAT and 42 (after minimised) in LTSA even if the parameter `N` is fixed to 3 and the input values from `in` is restricted to $\{0, 1\}$. The reason why the numbers of states are larger than one in Figure 11 is that they use standard semantics, thus variables must be instantiated to each value.

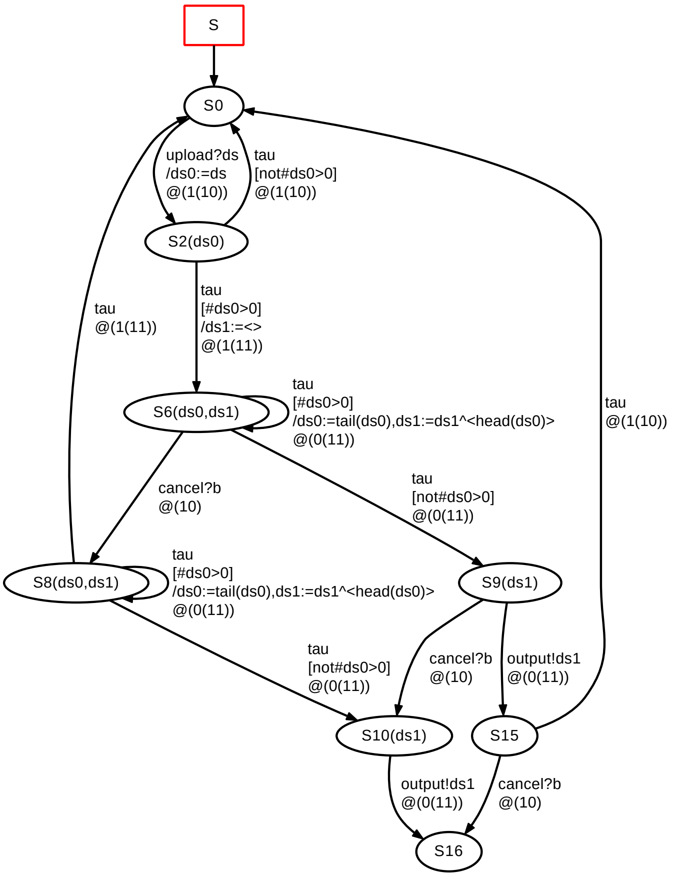


Figure 19. The reduced symbolic transition graph of $\mathtt{TransferSys}' \setminus \{| \, \mathtt{complete} \, |\}$

Li and Chen [9] presented an algorithm to translate the problem for checking bisimulation between symbolic transition graphs with assignment into the problem of solving a predicate equation system. The translation is sound and complete, but it is hard to automatically solve the generated predicate equation system.

Interactive theorem provers [20,21,22] for process algebra have been presented. In theorem provers, infinite state processes can be verified. It takes, however, time to make proof-scripts for giving proof-instructions. Especially, it is often necessary and difficult to manually give expected relations between a concurrent process and a sequential process. Probably, CONPASU can support to make such proof-script even for infinite state processes.

6. Conclusion

We have presented an analysis-method for reducing the number of states of the symbolic transition graphs based on a symbolic operational semantics with assignments and locations. It is guaranteed that the original process and the reduced process are stable-failures-equivalent. Then, we have implemented the symbolic operational semantics and the state-reduction method in the tool CONPASU, and demonstrated it. As far as we know, there is no other tool which can automatically generate symbolic transition graphs such as Figure 18 from concurrent processes such as Figure 15.

The sequential processes generated by CONPASU do not necessarily correspond to the expected ideal specifications. It is, however, often difficult to formally describe such ideal specifications. The generated sequential processes can give useful information for describing such ideal specifications.

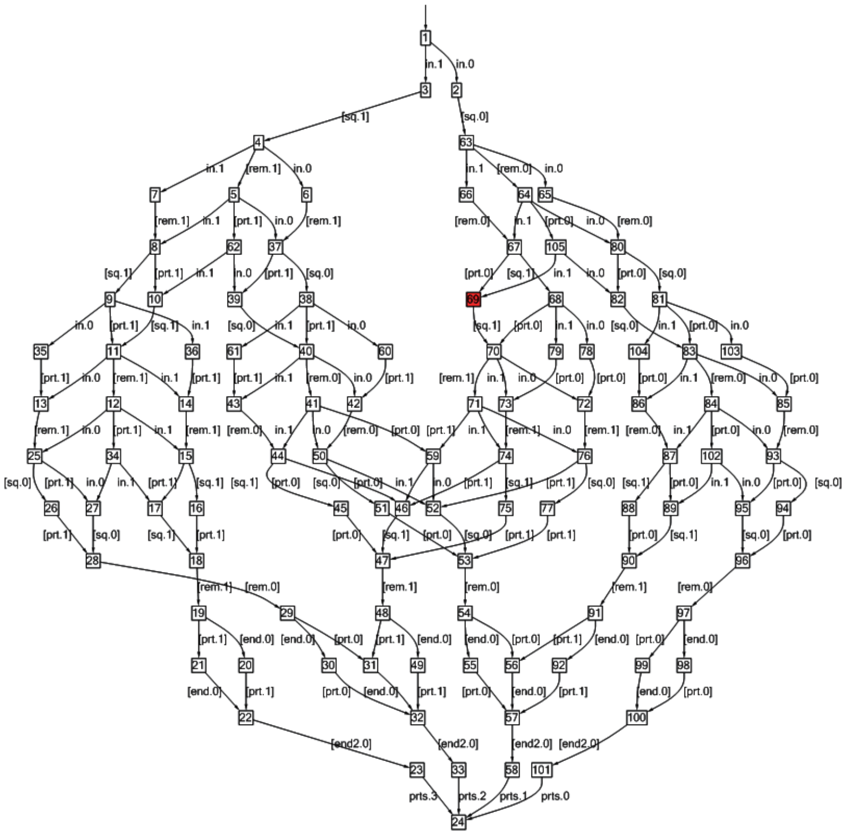


Figure 20. The transition graph of $\mathtt{CAL}(3)$ displayed by PAT (input-value $\in \{0, 1\}$)

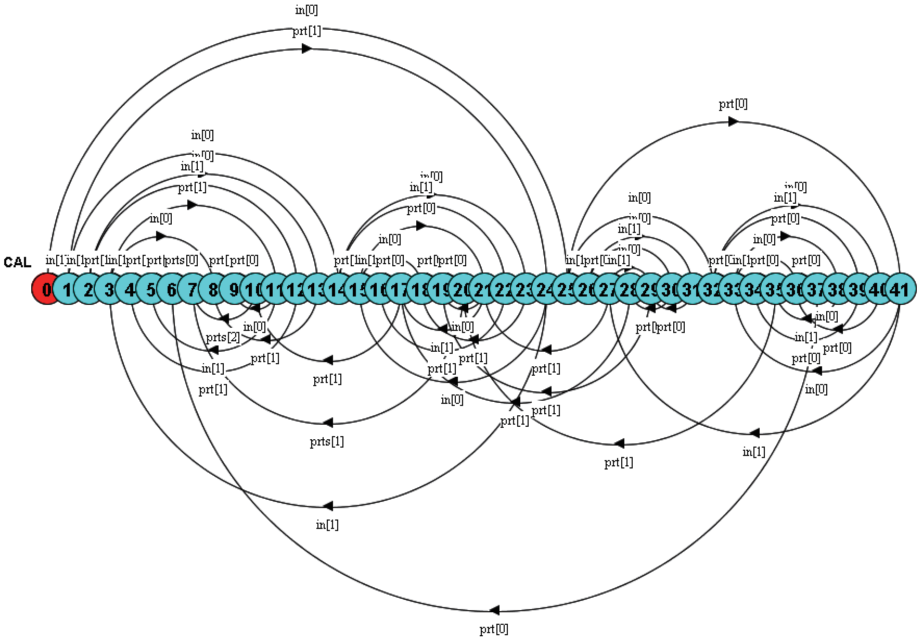


Figure 21. The minimised transition graph of $\mathtt{CAL}(3)$ displayed by LTSA (input-value $\in \{0, 1\}$)

The current CONPASU is a prototype and we have not discussed the performance of CONPASU yet. It is a future work to polish CONPASU and evaluate the performance. As a sample, it took 39 msec for computing the symbolically reducible set from the process in Figure 15 by Definition 2.7 and 46 msec for bypassing the process by Intel Core 2 Duo CPU P9600, 2.66 GHz, and 4 GB RAM. In the theoretical side, we are considering how bypass affects divergence. We have confirmed that divergence is not newly created in bypassed process of Definition 2.6, but we are still carefully discussing whether divergence can disappear by bypass or not. We conjecture that E and $(\mathtt{BP}^{(n)}_{(\mathcal{R},E)}(\tilde{x}))\backslash\{\mathtt{tmp}\}$ in Corollary 2.1 are also failures/divergence-equivalent.

Acknowledgments

This work was supported by JSPS-KAKENHI 20500023.

References

[1] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[2] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

[3] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[4] Formal Systems (Europe) Limited. Failures-divergence refinement: FDR2. `http://www.fsel.com/`.

[5] National University of Singapore. PAT: Process analysis toolkit. `http://www.comp.nus.edu.sg/~pat/`.

[6] The University of Edinburgh. The concurrency workbench. `http://homepages.inf.ed.ac.uk/perdita/cwb/`.

[7] Technische Universiteit Eindhoven. mcrl2. `http://www.mcrl2.org/mcrl2/wiki/index.php/Home`.

[8] M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138(2):353–389, 1995.

[9] Z. Li and H. Chen. Computing strong/weak bisimulation equivalences and observation congruence for value-passing processes. In *TACAS ’99*, LNCS 1579, pages 300–314. Springer-Verlag, 1999.

[10] H. Lin. Symbolic transition graph with assignment. In *CONCUR ’96*, LNCS 1119, pages 50–65. Springer-Verlag, 1996.

[11] R. S. Lazic. *A Semantic Study of Data Independence with Applications to Model Checking*. PhD thesis, Oxford University Computing Laboratory, 1999.

[12] U. Montanari and D. Yankelevich. A parametric approach to localities. In *ICALP ’92*, LNCS 623, pages 617–628. Springer-Verlag, 1992.

[13] G. Boudol, I. Castellani, M. Hennessy, and A. Kiehn. Observing localities. *Theoretical Computer Science*, 114:31–61, June 1993.

[14] R. Wimmer, M. Herbstritt, and B. Becker. Minimization of large state spaces using symbolic branching bisimulation. In *DDECS’06*, 2006.

[15] Y. Isobe. Webpage on CONPASU. `http://staff.aist.go.jp/y-isobe/conpasu/`.

[16] T. Nipkow, L. C. Paulon, and M. Wenzel. *Isabelle/HOL*. LNCS 2283. Springer, 2002.

[17] E. Gansner, E. Koutsofios, and S. North. Drawing graphs with dot, 2006. `http://www.graphviz.org/Documentation/dotguide.pdf`.

[18] J. Ellson, E. Gansner, E. Koutsofios, S. North, and G. Woodhull. Graphviz - graph visualization software. `http://www.graphviz.org/`.

[19] Imperial College London. LTSA - labelled transition system analyser. `http://www.doc.ic.ac.uk/ltsa/`.

[20] B. Dutertre and S. Schneider. Using a PVS embedding of CSP to verify authentication protocols. In *TPHOL 1997*, LNCS 1275, pages 121–136. Springer, 1997.

[21] Y. Isobe and M. Roggenbach. Webpage on CSP-Prover. `http://staff.aist.go.jp/y-isobe/CSP-Prover/CSP-Prover.html`.

[22] Y. Isobe and M. Roggenbach. A generic theorem prover of CSP refinement. In *TACAS 2005*, LNCS 3440, pages 108–123. Springer, 2005.