"ON-THE-FLY INSTANTIATION" OF VALUE-PASSING PROCESSES

Huimin Lin

Laboratory for Computer Science
Institute of Software, Chinese Academy of Sciences
lhm@ox.ios.ac.cn

Abstract: The traditional approach to automatic verification of value-passing processes is to translate them into pure processes, by instantiating input variables over all possible values, then submit the resulting pure processes to verification tools. The main disadvantage of this approach is that problems with infinite value domain can not be dealt with, even if these values are never used by the program, as in the case of communication protocols. In this paper we propose an algorithm which works directly on value-passing processes, represented as *symbolic transition graphs with assignment*. The key idea is to instantiate input variables "on-the-fly", while checking bisimulation, making it possible not to store the entire concrete transition system either before or during the verification process. As a consequence problems with large, even infinite, state spaces can be handled. The algorithm has been implemented and tested on some typical verification problems.

1 INTRODUCTION

The issue addressed in this paper is deciding bisimulation equivalences for value-passing concurrent systems. By "value-passing" we mean data values can be transmitted via communication channels between processes running in parallel. A typical class of value-passing processes is communication protocols. A characteristic feature of value-passing processes is that they can per-

Supported by research grants from the Chinese Academy of Sciences, the National Science Foundation of China, and EU KIT 119 project SYMSEM.

form input and output actions of the forms $c?x$ and $c!e$, where x is a data variable and e a data expression. In contrast, "pure" processes can only perform atomic actions which essentially allow processes to *synchronise* with each other, instead of exchange data. While algorithms and verification tools for pure processes have received considerable attention in the past two decades ([CPS93, SV89, GLZ89], among others), the area of automatic verification of value-passing processes remains almost unexplored.

The traditional approach to value-passing processes is to translate them into pure processes. A key step in such translation is to instantiate input variables with all possible values from the data domains which the variables range over. For example, the following defines a very simple value-passing process (in CCS syntax)

$$P = c?x.d!x.P$$

where x is a variable of type integer. All P can do is to repeatedly receive an integer along channel c then transmit it along channel d. Simple as it looks, so far there are no automatic verification tools that can handle it. To use existing tools one has to first translate P into pure process P':

$$P' = \ldots + c?{-1}.d!{-1}.P' + c?0.d!0.P' + c?1.d!1.P' + \ldots$$

which is also beyond the scope of the existing tools because P' has infinitely many transitions.

For value-passing processes there are two sensible notions of bisimulation, namely *early* and *late*, depending on when input variables get instantiated: during inferring transitions (early), or during matching for bisimulation (late) ([MPW92, HL95]). The translation-based approach can only handle early bisimulation.

In this paper we propose a new approach to value-passing systems without resorting to such translation. The approach is applicable to both late and early bisimulation equivalences. The main novelty is that it combines the *symbolic* framework with "on-the-fly" instantiation technique. As it has been recognised, in many practical value-passing systems (with communication protocols in particular) a distinction can often be made between *control* and *data* variables: the control variables may participate in complicated computations affecting the control flow of the systems, while the data variables just hold values "passively" without being tested nor computed. It is often the case that control variables take values from finite sets and data variables may assume values from infinite domains. For instance, in Alternating Bit Protocols the variables for the boolean flags are control variables, while those holding the values being transmitted are data variables.

We present an algorithm for deciding bisimulation equivalences between such value-passing processes. Instead of instantiating input variables before checking bisimulation, the algorithm instantiates these variables "on-the-fly", during the verification process. To see how it works, suppose we are comparing the following two processes for bisimulation:

$$c?x.P \qquad\qquad \text{and} \qquad\qquad c?y.Q$$

where x and y range over $\{1,2,3\}$. At this stage we know that the two processes can only mimic each other when x and y are instantiated with *the same value*. Therefore we can take each value v from the set $\{1,2,3\}$ in turn, substituting it for x in P and y in Q, then go on to compare $P[v/x]$ and $Q[v/y]$. There are only three pairs of residuals, namely $(P[1/x],\ Q[1/y])$, $(P[2/x],\ Q[2/y])$ and $(P[3/x],\ Q[3/y])$,to be compared at the next stage. In contrast, the translation-based approach would first translate them into

$$c?1.P[1/x]+c?2.P[2/x]+c?3.P[3/x] \ \text{ and } \ c?1.Q[1/x]+c?2.Q[2/x]+c?3.Q[3/x]$$

which will then require nine comparisons, with six definitely fail. If x and y are data variables, then we adopt the symbolic bisimulation technique by instantiating them with the least unused *symbolic values*.

To incorporate the on-the-fly instantiation strategy, as illustrated by the above example, the bisimulation checking algorithm should work on the process structures in "top-down" fashion. Such an algorithm is also called "on-the-fly" since, when working top-down, it is possible to generate the transition graphs piece-by-piece while checking bisimulation, instead of creating the whole transition graphs before verification starts [FM91]. For this idea to work the key issue is how to *efficiently* generate transitions "on-the-fly" at each stage. Direct calculating from process syntax using the operational semantics is too costly, because each time the same term is encountered the same calculation has to be repeated again, and due to recursion such repetition is unavoidable. Using the technique of "transition caching" may easy the pain of repeated calculations, but the caching table may take up a considerable amount of memory, a resource already in short supply.

The solution proposed in this paper is to use *symbolic transition graphs with assignment* (STGA for short). It is shown in [Lin96] that STGAs correspond exactly to *regular* value-passing processes, and moreover, they are closed under parallel composition, hence capable of representing *networks of parallel processes*. An example STGA is shown in Figure 1.1. As can be seen there, each node of a STGA is associated with a set of free data variables, and each edge carries three pieces of information: a boolean condition, a multiple assignment statement, and an abstract action. A *state* over a STGA, representing a *closed term*, then consists of a node together with a vector of data assigning values to the free variables at the node. These values can then be used to evaluate outgoing edges at the node, to produce concrete transitions, in the following manner: first the boolean condition is evaluated, if the result is *false* then this edge does not contribute to the real transition from this concrete state; Otherwise the assignment is evaluated and the result is used to update the data

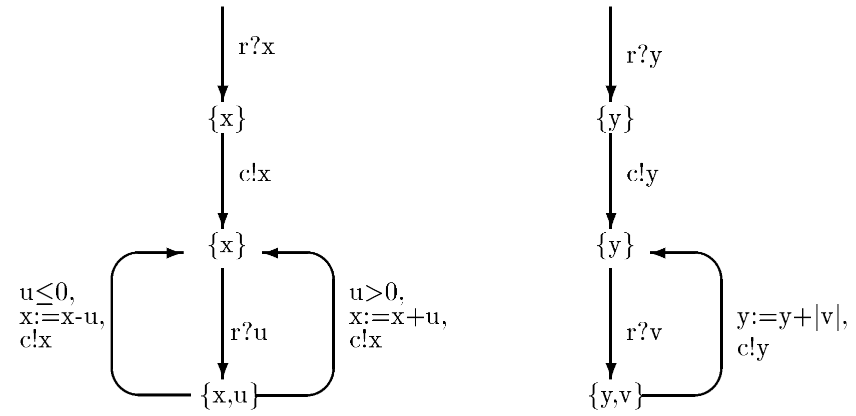

Figure 1.1 Example STGAs

values, resulting in a new vector of data used to evaluate the action if it is an output action (input actions are instantiated on-the-fly as described above).

STGAs can be generated from process terms using *symbolic transition semantics*. For regular processes the sizes of the graphs are linear to the length of the terms. In practice both the time used to generate STGAs and the amount of memory used to store them are negligible. STGAs can be regarded as "half-cooked" syntax: all process dynamics has been represented graphically, while data and boolean expressions remain "uncooked" and are carried in the edges. The task of deriving transitions from a closed process term is reduced to evaluating outgoing edges from a state.

A node in a STGA denotes a subterm, and edges leading to a common node represent sharing a common substructure. Thus STGAs allow to avoid not only direct manipulations on syntactical terms, but also repeated calculations due to recurrence of the same term.

Based on STGAs we have developed an algorithm exploiting the "on-the-fly instantiation" idea. Starting from the root nodes of two STGAs with initial value vectors, the algorithm tries to build the *smallest* bisimulation including the initial states. At each stage transitions are derived as described above. These transitions are then compared for bisimulation. The algorithm only stores pairs of state which have been compared. In particular, it does not store any concrete transitions (which requires more space than states). As a consequence it can handle problems with much larger state spaces than the partition algorithm which works on complete transition graphs. The algorithm has been implemented and tested on some typical verification problems, and the early results are very encouraging.

An important issue in designing verification algorithms and tools is the generation of informative diagnosis when verification fails, because producing just a "no" answer has little help in debugging the input. As mentioned before it is difficult for the the translation-based approach to generate helpful diagnosis information, due to the fact that most structuring information presented in the original value-passing processes gets lost in the translation. In this respect the on-the-fly instantiation algorithm has its advantage. When the two input value-passing processes are not bisimilar, it can generate diagnosis information in the form of a sequence of state pairs witnessing the failure. As a state consists of an (open) subterm in the process definition together with a list of values for the free data variables of the subterm, it is easy to debug the original design using such "high level" diagnosis information.

The rest of the paper is organised as follows: The notion of symbolic transition graphs with assignment is briefly reviewed in the next section. The algorithm is then presented in Section 1.4, with its correctness discussed and complexity analysed. Section 4 deals with early bisimulation. The paper is concluded with Section 5.

Related work The on-the-fly algorithm for deciding bisimulation for pure processes first appeared in [FM91] (to the author's knowledge). The Mobile Workbench also implemented a version of on-the-fly algorithm for open bisimulation in the context of π-calculus [VM94]. The main contribution of the present paper is twofold: *(1)* Lifting the on-the-fly bisimulation checking algorithm to handle value-passing processes, by instantiating input variables "on-the-fly" over symbolic transition graphs with assignment; *(2)* Combining the symbolic bisimulation and "on-the-fly instantiation" techniques so that a class of infinite-state problems can be dealt with.

2 AN ABSTRACT MODEL FOR VALUE-PASSING SYSTEMS

In this section we will only give a brief and informal introduction to *symbolic transition graphs with assignment*, and refer the readers to [Lin96] for a detailed account on this subject.

We assume the existence of the following sets: *Val* (data values), ranged over by $v, u, \ldots$; *Var* (data variables), ranged over by $x, y, z, \ldots$; *DExp* (data expressions), ranged over by $e, e', \ldots$; *BExp* (boolean expressions), ranged overy by $b, b', \ldots$; *Chan* (channel names), ranged overy by $c, c', \ldots$. We will use the customary notations $\overline{x} := \overline{e}$ for multiple assignment, $[\overline{e}/\overline{x}]$ for substitution, and sometime identify $\overline{x} := \overline{e}$ with $[\overline{e}/\overline{x}]$. It is assumed that $\mathit{Var} \cup \mathit{Val} \subseteq \mathit{DExp}$, and $e = e' \in \mathit{BExp}$ for any $e,\ e' \in \mathit{DExp}$. We also assume that data expressions are equipped with reasonable notions of free and bound variables, and write $\mathit{fv}(e)/\mathit{bv}(e)$ for the sets of free/bound variables of e.

The set of abstract actions, $\{\, \tau, c?x, c!e \mid c \in \mathit{Chan}, x \in Var, e \in \mathit{DExp}\,\}$, is ranged over by α. The sets of free/bound variables of abstract actions are

$$\frac{m \overset{b,\theta,\tau}{\longmapsto} n}{m_\rho \xrightarrow{\tau} n_{\theta\rho}} \ \rho \models b \qquad \frac{m \overset{b,\theta,c!e}{\longmapsto} n}{m_\rho \xrightarrow{c!\rho(e\theta)} n_{\theta\rho}} \ \rho \models b \qquad \frac{m \overset{b,\theta,c?y}{\longmapsto} n}{m_\rho \xrightarrow{c?y} n_{\theta\rho}} \ \rho \models b$$

Figure 1.2 Late Operational Semantics

defined by $fv(c!e) = fv(e)$, $bv(c?x) = \{x\}$, and $fv(\alpha) = bv(\alpha) = \emptyset$ for any other action α.

An *evaluation* $\rho \in Eval$ is a type-respecting mapping from Var to Val and we use the standard notation $\rho\{x \mapsto v\}$ to denote the evaluation which differs from ρ only in that it maps x to v. An application of ρ to a data expression e, denoted $\rho(e)$, always yields a value from Val and similarly for boolean expressions; $\rho(b)$ is either true or false. We will write $\rho \models b$ to indicate that $\rho(b) = true$. If $\sigma \equiv [\overline{e}/\overline{x}]$ then the application of σ to ρ is denoted by $\sigma\rho = \rho\{\overline{x} \mapsto \rho(\overline{e})\}$.

Definition 2.1 A symbolic transition graph with assignments (STGA for short) is a rooted directed graph where each node n is associated with a finite set of free variables $fv(n)$ and each edge is labeled by a triple $(b, \overline{x} := \overline{e}, \alpha)$. A STGA is *well-formed* if for each edge from n to m, written $n \overset{b,\overline{x}:=\overline{e},\alpha}{\longmapsto} m$, it holds that $fv(b, \overline{e}) \subseteq fv(n)$, $fv(\alpha) \subseteq \{\overline{x}\}$, and $fv(m) \subseteq \{\overline{x}\} \cup bv(\alpha)$. $\square$

We will often simply write $n \overset{\theta,\alpha}{\longmapsto} m$ for $n \overset{true,\theta,\alpha}{\longmapsto} m$, and will also omit θ when it is the identity assignment on $fv(n)$.

Traditionally (pure) processes are modeled by *labeled transition systems* (LTSs for short) which are directed graphs in which arcs are labeled with atomic actions. A vertex in a LTS represents a state and the outgoing arcs show the possible actions the process at the state can perform to evolve into new states. Existing verification algorithms and automatic tools for process algebras are all based on LTSs ([CPS93, SV89, GLZ89]).

STGAs are graphic representations for value-passing processes, just as traditional transition graphs for pure processes. However the STGA representations are at a very abstract level as most of the characteristic structures of value-passing processes, namely data expressions, boolean conditions and input/output actions, are retained in the edges.

A STGA $\mathcal{G}$ can be "expanded" into a "partial" concrete labeled transition system whose vertices are pairs of (n, ρ) with n a node of $\mathcal{G}$ and ρ an evaluation mapping free variables at n to values. Transitions are generated according to the operational semantics given in Figure 1.2. Transition systems so generated are called "partial" because input actions are not yet instantiated. The instantiation will be carried out while graphs are compared for bisimulation.

Definition 2.2 A late bisimulation is a symmetric relation R over states such that: if $(m_\rho, n_\varrho) \in R$ then

1. $m_\rho \stackrel{c?y}{\longrightarrow} m'_{\rho'}$ implies there exists $n_\varrho \stackrel{c?z}{\longrightarrow} n'_{\varrho'}$ and for all $v \in \mathit{Val}$ $(m'_{\rho'\{y\mapsto v\}}, n'_{\varrho'\{z\mapsto v\}}) \in R$.

2. for any other actions $m_\rho \stackrel{a}{\longrightarrow} m'_{\rho'}$ implies there exists $n_\varrho \stackrel{a}{\longrightarrow} n'_{\varrho'}$ and $(m'_{\rho'}, n'_{\varrho'}) \in R$.

We write $m_\rho \sim n_\varrho$ if there is a late bisimulation R such that $(m_\rho, n_\varrho) \in R$. Two graphs $\mathcal{G}$ and $\mathcal{G}'$ are bisimilar with respect to ρ and ρ' if $r_\rho \sim r'_{\rho'}$ where r and r' are the roots of the two graphs, respectively. $\square$

As an example of how to generate symbolic graphs from a process description language let us consider regular value–passing *CCS* given by the following BNF grammar:

$$t \quad ::= \quad \mathbf{0} \mid \alpha.t \mid b \to t \mid t + t \mid P(\overline{e})$$

where for each process identifier P there is a definition $P(\overline{x}) \Leftarrow t$ which satisfies $fv(t) \subseteq \{\overline{x}\}$ and is *guarded*, *i.e.* every identifier in t is within the scope of an action prefixing $\alpha.\_$. Given a term t in this language a STGA can be generated using the following rules:

$$\frac{}{\alpha.t \stackrel{true,\emptyset,\alpha}{\longmapsto} t} \qquad \frac{t \stackrel{b,\theta,\alpha}{\longmapsto} t'}{t + u \stackrel{b,\theta,\alpha}{\longmapsto} t'} \qquad \frac{t \stackrel{b,\theta,\alpha}{\longmapsto} t'}{u + t \stackrel{b,\theta,\alpha}{\longmapsto} t'}$$

$$\frac{t \stackrel{b,\theta,\alpha}{\longmapsto} t'}{b' \to t \stackrel{b\wedge b',\theta,\alpha}{\longmapsto} t'} \qquad \frac{t \stackrel{b,\theta,\alpha}{\longmapsto} t'}{P(\overline{e}) \stackrel{b[\overline{e}/\overline{x}],\theta[\overline{x}:=\overline{e}],\alpha}{\longmapsto} t'} \quad P(\overline{x}) \Leftarrow t \text{ is a definition}$$

Two STGAs can be composed using parallel composition operator $\|$ and restriction operator $\backslash$. When composing two STGAs it is required that they use disjoint name spaces for data variables (so the only way for them to cooperate is through communication). The nodes of the parallel composition $(\mathcal{G}\|\mathcal{H})\backslash R$, with R a set of channel names, are pairs of those of $\mathcal{G}$ and $\mathcal{H}$ with $fv(<n,m>) = fv(n) \cup fv(m)$, and the edges are created by the following rules (where the symmetric rules of par and com have been omitted):

$$\text{par} \ \frac{n \stackrel{b,\overline{x}:=\overline{e},\alpha}{\longmapsto} n'}{<n,m> \stackrel{b,\overline{x},\overline{y}:=\overline{e},\overline{y},\alpha}{\longmapsto} <n',m>} \quad \begin{array}{l} chan(\alpha) \cap R = \emptyset \\ fv(m) = \{\overline{y}\} \end{array}$$

$$\text{com} \ \frac{n \stackrel{b_1,\overline{x}:=\overline{e}_1,c?z}{\longmapsto} n', \qquad m \stackrel{b_2,\overline{y}:=\overline{e}_2,c!\overline{e}}{\longmapsto} m'}{<n,m> \stackrel{b_1\wedge b_2,\overline{x},\overline{y},z:=\overline{e}_1,\overline{e}_2,e[\overline{e}_2/\overline{y}],\tau}{\longmapsto} <n',m'>}$$

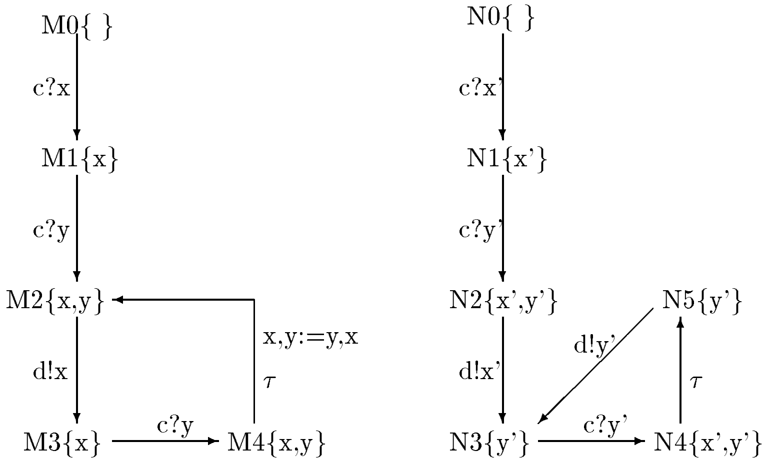

Figure 1.3

It is easy to see that parallel composition is symmetric and associative. This is CCS-style "handshaking" communication where communication happens between two processes capable of performing complementary (input and output) actions, yielding an "invisible" τ action. Similarly one can define parallel composition adopting "broadcasting" communication, as in CSP and LOTOS.

3 THE ALGORITHM

Given a symbolic graph, a variable is "data-independent" if it does not appear in the boolean condition part of any edge label, nor occur as a proper subterm of any data expression. Thus data variables are not subject to any operation other than assignment. The algorithm presented in this section works on symbolic graphs where all variables are either data-independent or with finite domains.

As observed in [JP92], it is sufficient to "instantiate" each data variable with a fresh *schematic variable* when deciding bisimulation. The approach proposed in [JP92] is based on the (global) partition algorithm and requires *a prior* construction of complete transition systems. Here we incorporate the idea by adopting the "symbolic bisimulation" technique [HL95]. We assume a countable set of "symbolic values" which are totally ordered, and the function *nextSVal*(s, t) always returns the smallest symbolic value that does not appear in the value vectors at states s and t. To illustrate how symbolic values are generated, we use the example shown in Figure 1.3 where all variables are data-independent. Assume the symbolic values are $v_1, v_2, v_3, \ldots$ (in that order). We start with the root nodes. At the state pair

$(M0\{\},\ N0\{\})$ no symbolic value is used, hence $nextSVal(M0\{\},\ N0\{\}) = v_1$ which is associated to x and x'. Now at the new state pair $(M1\{x = v_1\},\ N1\{x' = v_1\})$ we have $nextSVal(M1\{x = v_1\},\ N1\{x' = v_1\}) = v_2$, so the next state pair is $(M2\{x = v_1, y = v_2\},\ N2\{x' = v_1, y' = v_2\})$. After the output transitions the new state pair is $(M3\{x = v_1\},\ N3\{y' = v_2\})$. Since $nextSVal(M3\{x = v_1\},\ N3\{y' = v_2\}) = v_3$, the next state pair is $(M4\{x = v_1, y = v_3\},\ N3\{y' = v_3\})$. Passing through the τ and output transitions we go back to to nodes $M3$ and $N3$ again, with the state pair $(M3\{x = v_3\},\ N3\{y' = v_3\})$ (note that x and y got transposed after τ move). Now we have $nextSVal(M3\{x = v_3\},\ N3\{y' = v_3\}) = v_1$.

The algorithm for deciding late bisimulation for symbolic transition graphs with assignment is presented in Figure 1.4. It operates on a pair of STGAs $\mathcal{G}$ and $\mathcal{H}$. The function $bisim(s,t)$ will always terminate with answer *true* or *false*. It starts with the initial states pair (s,t), trying to find the smallest bisimulation containing the pair by matching transitions from each pair of states it reaches. While searching the symbolic graphs, at each pair of nodes the algorithm uses the values for the state variables to evaluate the edges, according to the operational semantics in Figure 1.2, producing concrete transitions and next states. The transitions are then matched for bisimulation, and the algorithm goes on to the new state pairs if the matches are successful. However when input edges are encountered the evaluation can not be fully carried out because the values for the input variables at the next pair of nodes are not yet known. In this case the algorithm behaves as follows: if the input variables are data variables then the function *nextSVal*(s, t) is called and the returned symbolic value is used for the input variables; otherwise each element is taken from the value domain associated with the input variables, one by one, to instantiate the input variables "on-the-fly". The algorithm then moves to the new states.

Three auxiliary data structures are used:

- *NotBisim*: List of state pairs which have already been detected as not bisimilar.
- *Visited*: List of state pairs which have already been visited.
- *Assumed*: List of state pairs which have already been visited and assumed to be bisimilar.

The core function, *match*, is called from function *bis*inside the main function *bisim*. It performs a depth-first search on the product of the two concrete transition graphs which are never fully created. Instead states and transitions of the concrete graphs are only generated from the symbolic graphs on-the-fly, when needed. Whenever a new pair of states is encountered it is inserted into *Visited*. If two states fail to match each other's transitions then they are not

$bisim$(s, t) = {
    $NotBisim$:= { }
    fun bis(s, t) = {
        $Visited$:= { }
        $Assumed$:= { }
        $match$(s, t)}
    } handle $WrongAssumption \Rightarrow bis$(s, t)
    return(bis(s, t))

$match$(s, t) =
    $Visited := \{(s,\ t)\} \cup Visited$
    b = $\bigwedge_{a \in \{\tau, c!v, c?\}} match_a(s,\ t)$
    if b = *false* then {
        $NotBisim := \{(s,\ t)\} \cup NotBisim$
        if $(s,\ t) \in Assumed$ then raise $WrongAssumption$}
    return(b)

$match_{\{\ a | a \in \{\tau, c!v\}\ \}}$(s, t) =
    for each $s \xrightarrow{a} s_i$, for each $t \xrightarrow{a} t_j$
        $b_{ij} = close(s_i,\ t_j)$
    return$((\bigwedge_i(\bigvee_j b_{ij})) \wedge (\bigwedge_j(\bigvee_i b_{ij})))$

$match_{c?}$(s, t) =
    for each $s \xrightarrow{c?x} s_i$, for each $t \xrightarrow{c?y} t_j$
        if x and y are data-independent then
            let $z = nextSVal(s,\ t)$
                $b_{ij} = close(s_i[x \mapsto z],\ t_j[y \mapsto z])$
            return$((\bigwedge_i(\bigvee_j \forall z b_{ij})) \wedge (\bigwedge_j(\bigvee_i \forall z b_{ij})))$
        else for each $s \xrightarrow{c?x} s_i$, for each $t \xrightarrow{c?y} t_j$
            for each $v \in Val$
                $b^v_{ij} = close(s_i[x \mapsto v],\ t_j[y \mapsto v])$
            return$((\bigwedge_i(\bigvee_j \bigwedge_{v \in Val} b^v_{ij})) \wedge (\bigwedge_j(\bigvee_i \bigwedge_{v \in Val} b^v_{ij})))$

$close(s,\ t)$ =
    if $(s,\ t) \in NotBisim$ then return(*false*)
    else if $(s,\ t) \in Visited$ then {
        $Assumed := \{(s,\ t)\} \cup Assumed$
        return(*true*)}
    else return($match$(s, t))

Figure 1.4 The algorithm

bisimilar and the pair is inserted into *NotBisim*. In case the current state pair has been visited before, we first check if it is in *NotBisim* and return *false* if so. Otherwise a loop has been detected and we make assumption that the two states are bisimilar, by inserting the pair into *Assumed*, and return *true*. If we find the two states are not bisimilar after finishing searching the loop, then we know that the assumption is wrong, so we first add the pair into *NotBisim* and then raise the exception *WrongAssumption* which forces a rerun of *bis*, with one piece of new information, namely the two states in this pair are not bisimilar.

The correctness of the algorithm is not difficult to justify. Suppose there are k data-independent variables at nodes n and m, then, at most $k+1$ different schematic values will be used in any state pair $(s,\ t)$ at $(n,\ m)$. So termination is guaranteed. For these variables the only places they can be used are simple output actions of the form $c!x$ where x is data-independent. When the algorithm returns *true* any pair of such output actions successfully passed the comparison in $match_{c!}$ must have their variables instantiated with the same schematic value, *i.e.* they are instantiated at the same time, when the two input actions which bind the output variables are compared in $match_{c?}$. Therefore if the input variables are instantiated with the same "real" value, the two output actions will still match each other.

Each call of *bis*(s,t) performs a depth-first search in the product graph of the two concrete transition graphs (which are "generated" piece by piece from the symbolic graphs during the search). *bis*(s,t) is only recalled when the exception *WrongAssumption* has been raised (by *match*, as explained above), in this case the size of *NotBisim* has been increased by at least one. Hence *bis*(s,t) can only be called for finitely many times. Therefore *bisim*(s,t) will always terminate. If *bisim*(s,t) terminates with *true*, then the set (*Visited* $-$ *NotBisim*) constitutes a bisimulation containing (s,t):

Theorem 1 *Suppose G and H are finite symbolic graphs with roots n and m and initial values ρ and ρ', respectively, and the value domains for the control variables are finite. Function $bisim((n,\rho),(m,\rho'))$ always terminates, and if it returns true then (n,ρ) and (m,ρ') are bisimilar.*

On the other hand, when the algorithm returns *false* it is not necessary that the two graphs are not bisimilar when data-independent variables are interpreted over *any* domain. This can be illustrated by a very simple example: Running the algorithm on the two processes

$$c?x.c?y.d!x.d!y.\mathbf{0} \qquad \text{and} \qquad c?x.c?y.d!y.d!x.\mathbf{0}$$

will return *false*, but they are bisimilar when $x,\ y$ are only allowed to take value from a singleton set. Intuitively this is because the data domain involved does not have enough elements to distinguish between different data-independent variables. To give a precise characterisation let $DIN_{\mathcal{G}}$ be the largest number of data-independent variables at any node of graph $\mathcal{G}$, and $DIN_{\{\mathcal{G},\mathcal{H}\}}$ the minimum of $DIN_{\mathcal{G}}$ and $DIN_{\mathcal{H}}$. Then

Proposition 3.1 *If the algorithm returns false then the two input processes, with symbolic graphs $\mathcal{G}$ and $\mathcal{H}$, are not bisimilar when their data-independent variables take values over any data domain with cardinality no less than $DIN_{\{\mathcal{G},\mathcal{H}\}}$.*

To see the complexity of the algorithm, let N be the number of nodes in the production graph of the two concrete transition graphs. The time required for the first call of *bis* is (at most) $O(N)$, for the second call is $O(N-1)$, ... Hence the worst case complexity of *bisim* is $O(N+(N-1)+...+1) = O(N^2)$. However experiences show that the square fact does not appear in practical applications: in most cases the algorithm terminates with only one or two calls of *bis*.

The algorithm has been implemented in Standard ML, and tested on a number of examples. In the Appendix is the input file for the alternating-bit protocol problem which allows the media to lose data (adapted from an example in pure CCS included in the Concurrency Workbench distribution). In the file a type named `message` is defined as a subtype of integer with range `1 ... 10`. Then all process identifiers, channel names and variables are declared with appropriate types (`Bool` is a built-in type), followed by the conjecture to prove. Finally, recursive definitions are provided for the process identifiers (the `where` part). The right hand-side of the conjecture is the specification of the protocol, while the left hand-side is the implementation. Both the specification and the implementation are compiled into STGAs before subject to the algorithm. The STGA for the specification has only 2 nodes and 2 edges, while the one for the implementation has 32 nodes and 80 edges.

As comparison we also run the same examples using the Concurrency Workbench of North Carolina [CS96] which implements the partition algorithm for deciding bisimulation [KS83]. As the examples are written in full CCS Bruns' value-passing front-end [Bru96] is used to translate them into pure CCS before fed into the Workbench. Both systems are written in New Jersey ML version 109, and all results are obtained on a Sparc station with 60MHZ twin-CPU and 64MB memory. The first example is the alternating-bit protocol mentioned above. The second is the "triple-modular redundancy" problem taken from [Bru96]. By increasing the sizes of data domains bigger and bigger transition graphs can be achieved. When the domains grow up to certain limits (100 in the case of alternating-bit protocol and 20 in the case of triple modular redundancy problem) the Workbench runs out of memory and stops. On the other hand the on-the-fly algorithm can handle much larger domains: 350 for the alternating-bit protocol (with 32204 states) and 140 for the triple modular redundancy problem (with 140561 states). Moreover, the variables holding the messages being transmitted in the alternating-bit protocol are data independent. When it is declared as such (by changing the definition of type `message` to `message = data` in the input file) the verification takes less than a second to finish with answer *true*, which means the protocol is correct over even infinite data domains.

When the input processes are not bisimilar, the algorithm terminates with *false*. In this case informative diagnosis information can be easily computed from *NotBisim*. It consists of a sequence of matching transition and ends with a pair of states, one of which can do a transition that the other can not. For example, if we change the definition of process R(rb) in the AB-protocol example in the Appendix to

```
R(rb) = tau.sack!(not(rb)).R(rb) +
        r?(ra,rm).if ra==rb then receive!rm.sack!ra.R(rb)
                  else sack!ra.R(rb)
```

i.e., when the newly received frame has expected boolean flag (`ra==rb`), the message body is transmitted (`receive!rm`) and an acknowledge is sent back (`sack!ra`), but the flag is wrongly remained unswitched (`R(rb)` instead of `R(not(rb))` as in the original description). Now feed the revised file to the algorithm, we will get the following diagnosis:

```
<R(rb)|Mlossy|S(sb){rb=false,sb=false},Spec{}>==send?sm,send?m=>
<R(rb)|Mlossy|S1(sb,sm){rb=false,sb=false,sm=1},
    receive!m.Spec{m=1}>==receive!1,receive!1=>
<R(rb)|Mlossy|if sa==sb then S(not(sb)) else S1(sb,sm)
    {rb=false,sa=false,sb=false,sm=1},Spec{}>==send?sm,send?m=>
<R(rb)|Mlossy|S1(sb,sm){rb=false,sb=true,sm=1},
    receive!m.Spec{m=1}>

now <receive!m.Spec{m=1}>==receive!1=><Spec{}>
but <R(rb)|Mlossy|S1(sb,sm){rb=false,sb=true,sm=1}> has no
matching move
```

The last two lines read: the specification is at a state where a transmission is possible (`receive!1`), while the implementation is at the state `R(rb)|Mlossy|S1(sb,sm)` with boolean flags `rb=false, sb=true` and can not perform such a transition (in fact it is ready to do a `send?sm` action).

Such diagnosis information points directly to the source-level process description, hence is very helpful for debugging purpose.

4 THE EARLY CASE

The bisimulation equivalence discussed above is called "late" because input variables are instantiated when input actions are matched for bisimulation, instead of when concrete transitions are inferred from process terms (i.e. when input edges are evaluated in our case) which takes place before bisimulation matching. In late bisimulation each abstract input action of the form $c?x$ from one process has to be matched by a *single* corresponding action from the other, for all possible values. If this restriction is relaxed by allowing an input action to be matched by several input actions, possibly instantiated with different values, from the other process, the resulting relation is called "early bisimulation".

Early bisimulation can be obtained by swapping the quantifications "for all $v \in Val$" and "there exists" in the first clause of Definition 2.2, so for different values the input move from m_ρ is allowed to be matched by different input moved from n_ϱ:

1. $m_\rho \xrightarrow{c?y} m'_{\rho'}$ implies for all $v \in Val$ there exists $n_\varrho \xrightarrow{c?z} n'_{\varrho'}$ such that $(m'_{\rho'\{y \mapsto v\}}, n'_{\varrho'\{z \mapsto v\}}) \in R$.

Concerning the algorithm only the $match_{c?}$ function needs modifying:

$$
\begin{array}{l}
match^{E}_{c?}(\text{s, t}) = \text{for each } s \xrightarrow{c?x} s_i \\
\qquad \text{if } x \text{ is data-independent then} \\
\qquad\qquad \text{let } z = nextSVal(s,\ t) \\
\qquad\qquad\quad \text{for each } t \xrightarrow{c?y} t_j \\
\qquad\qquad\qquad b_{ij} = close(s_i[x \mapsto z],\ t_j[y \mapsto z]) \\
\qquad\qquad \text{return}((\bigwedge_i (\forall z \bigvee_j b_{ij})) \wedge (\bigwedge_j (\forall z \bigvee_i b_{ij}))) \\
\qquad \text{else for each } v \in Val \\
\qquad\qquad\quad \text{for each } t \xrightarrow{c?y} t_j \\
\qquad\qquad\qquad b^v_{ij} = close(s_i[x \mapsto v],\ t_j[y \mapsto v]) \\
\qquad\qquad \text{return}((\bigwedge_i (\bigwedge_{v \in Val} (\bigvee_j b^v_{ij}))) \wedge (\bigwedge_j (\bigwedge_{v \in Val} (\bigvee_i b^v_{ij}))))
\end{array}
$$

5 CONCLUSIONS

We have presented new algorithms for deciding both late and early bisimulation equivalences for value-passing processes. The main novelty of our approach is that input actions are instantiated on the fly, making it possible to check bisimulation for value-passing processes without first expanding them into pure processes. Process terms are compiled into *symbolic transition graphs with assignment*. During verification concrete transitions are derived from the symbolic graphs hence there is no need to store entire concrete transition systems, in particular transitions are never stored. The technique of symbolic bisimulation has been incorporated so that data-independent variables ranging over potentially infinite value domains can also be dealt with. The algorithm has been implemented and tested on some examples, and the results show that it can handle much larger problems than the translation-based approach. When the input graphs are not bisimilar an informative ("source-level") diagnosis information can be generated, which is very helpful for debugging purpose.

It should be pointed out that the on-the-fly instantiation method does not rely on any particular properties of the data domain involved, hence any techniques for reducing data domain size can be incorporated.

As further work we are adapting the "on-the-fly instantiation" algorithm to check early and late bisimulations for the π-calculus, which can be viewed as complementary to the *Mobile Workbench* [VM94], a verification tool dedicated to checking *open bisimulation* in the π-calculus.

References

[Bru96] G. Bruns. *Distributed Systems Analysis with CCS.* Prentice Hall, 1996.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. A semantics based verification tool for the verification of concurrent systems. *ACM Trans. Program. Lang. Syst.*, 15(1):36–72, 1993.

[CS96] R. Cleaveland and S. Sims. The NCSU Concurrency Workbench. In *CAV'96*, number 1102 in LNCS, pages 394 – 397. Springer–Verlag, 1996.

[FM91] J.-C. Fernandez and L. Mounier. On the fly verification of behavioural equivalences and preorders. In *CAV'91*, volume 575 of *LNCS*, pages 181–191. Springer–Verlag, 1991.

[GLZ89] J. Godskesen, K. Larsen, and M. Zeeberg. Tav user manual. Report R89-19, Aalborg University, 1989.

[HL95] M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138:353 – 389, 1995.

[JP92] B. Jonsson and J. Parrow. Deciding bisimulation equivalences for a class of non-finite-state programs. *Information and Computation*, 1992.

[KS83] P. Kanellakis and S. Smolka. CCS expressions, finite state processes, and three problems of equivalence. In *Proc. 2nd Ann. ACM Symposium on Principles of Distributed Computing*, pages 228–240, 1983.

[Lin96] H. Lin. Symbolic transition graph with assignment. In *CONCUR'96*, volume 1119 of *LNCS*, pages 50 – 65. Springer–Verlag, 1996.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile proceses, part I,II. *Information and Computation*, 100:1–77, 1992.

[SV89] R. De Simone and D. Vergamimi. Aboard auto. Report RT111, INRIA, 1989.

[VM94] B. Victor and F. Moller. The mobility workbench – a tool for the π-calculus. In *CAV'94*, number 818 in LNCS, pages 428 – 440. Springer–Verlag, 1994.

Appendix: An Example Input File

```
type
        message = 1 ... 10
process
        S : Bool
        S1 : Bool message
        S2 : Bool message
        Mlossy :
        Msafe :
        R : Bool

        Spec :
channel
        send : message
        receive : message
        r : Bool message
        s : Bool message
        rack : Bool
        sack : Bool
variable
        sb,sa,rb,ra,ma : Bool
        sm,rm,mm,m : message
conjecture

        (R(false) | Mlossy | S(false))\{r,s,rack,sack} = Spec
where

 S(sb) = send?sm.S1(sb,sm)
 S1(sb,sm) = s!(sb,sm).S2(sb,sm)
 S2(sb,sm) = tau.S1(sb,sm) + rack?sa.(if sa==sb then S(not(sb))
                                       else S1(sb,sm))

 R(rb) = tau.sack!(not(rb)).R(rb) +
         r?(ra,rm).if ra==rb then receive!rm.sack!ra.R(not(rb))
                   else sack!ra.R(rb)

 Mlossy = s?(ma,mm).(r!(ma,mm).Mlossy + Mlossy) +
          sack?ma.(rack!ma.Mlossy + Mlossy)

 Spec = send?m.receive!m.Spec

end
```