Local and Symbolic Bisimulation using Tabled Constraint Logic Programming*

Samik Basu[1]
Madhavan Mukund[2]
C.R. Ramakrishnan[1]
I.V. Ramakrishnan[1]
Rakesh Verma[3]

[1] Dept. of Computer Science, SUNY at Stony Brook, New York.
[2] Chennai Mathematical Institute, Chennai, India.
[3] Dept. of Computer Science, University of Houston, Texas.

E-mail : bsamik@cs.sunysb.edu, madhavan@smi.ernet.in, cram@cs.sunysb.edu, ram@cs.sunysb.edu, rmverma@cs.uh.edu

Abstract

Bisimulation is a fundamental notion that characterizes behavioral equivalence of concurrent systems. In this paper, we study the problem of encoding efficient bisimulation checkers for finite- as well as infinite-state systems as logic programs. We begin with a straightforward and short (less than 10 lines) encoding of finite-state bisimulation checker as a tabled logic program. In a goal-directed system like XSB, this encoding yields a *local* bisimulation checker: one where state space exploration is done only until a dissimilarity is revealed. Local checking can often outperform the traditional global checking by several orders of magnitude even for finite-state systems, as our experimental results show. Surprisingly, even when the systems are equivalent where the entire state space may need to be explored, the performance of our local checker is comparable to hand-coded equivalence checking algorithms implemented in other verification tools.

More importantly, the logic programming formulation of local bisimulation can be extended to do *symbolic bisimulation* for checking the equivalence of infinite-state concurrent systems represented by *symbolic transition systems*. We show how the two variants of symbolic bisimulation (late and early equivalences) can be formulated as a tabled constraint logic program in a way that precisely brings out their differences. We use a a constraint meta-interpreter over disequality constraints that evaluates tabled constraint logic programs to support the symbolic bisimulation checker. We present experimental results to illustrate the practical efficacy of our logic programming based symbolic bisimulation checker. Finally, we show that our symbolic bisimulation checker, despite the overheads imposed by the constraint meta-interpreter, actually outperforms non-symbolic checkers even for relatively small finite-state systems.

*This research was partially supported by NSF Grants CDA-9805735, EIA-9705998, CCR-9876242, and CCR 9732186.

1 Introduction

A tabled logic programming system offers an attractive platform for encoding computational problems in the specification and verification of systems. The XMC system [12] casts the problem of *model checking*— verifying whether a given concurrent system is in the model of a temporal logic formula— as query evaluation over an "equivalent" logic program [11]. This formulation is based on the connection between models of logic programs and models of temporal logics. In this paper, we consider the related problem of *bisimulation checking* which checks for equivalence of two system descriptions.

Bisimulation checking is a problem of fundamental importance in verification. Many verification systems such as the Concurrency Workbench of the New Century (CWB-NC) [3] and CADP [1] incorporate bisimulation checkers in their tool sets. Informally, a pair of automata M, M' are said to be bisimilar if for every transition in M there exists a corresponding transition in M' and vice versa. There has been a lot of research on efficient algorithms for bisimulation checking. But the focus of this vast body of work has been on finite-state systems, i.e., one assumes that M and M' are both finite state. But in many practical problems that arise in verification M and M' are no longer finite-state systems. Hennessy and Lin were the first to consider the problem of bisimilarity checking of infinite-state systems [4] in the setting of value-passing languages. This initial work has been recently expanded [7, 6]. Nevertheless research on this problem remains in a state of infancy.

In this paper, we explore the use of logic programming for the above problem. We begin with a direct formulation of strong- and weak-bisimulation checking for finite-state systems (see Section 2). We show that, using query evaluation with a tabled logic programming system, this encoding yields a *local* bisimulation checker: one where the state space of the concurrent systems is explored only until the first evidence of non-bisimilarity is found. Note that when the systems are indeed bisimilar, the local checker explores the entire (reachable) state space. Even in this case, our bisimulation checker encoded in XSB logic programming system [13] shows performance comparable to the global bisimulation checker in CWB-NC. For systems that are non-bisimilar, the local checker outperforms the global checker by several orders of magnitude.

Having established the baseline that logic-programming-based bisimulation checkers can indeed be practical even for finite-state systems, we consider the problem of checking equivalence of infinite-state systems. In particular, we introduce *symbolic transition systems* (STSs) which can finitely represent infinite-state systems (see Section 3). STSs are more general than Symbolic Transition Graphs (STGs) and STGs with Assignments (STGAs) used in [4] and [7] respectively. We formulate symbolic bisimulation algorithms for checking two kinds of equivalences widely studied in the literature— *late*- and *early*-equivalences— as tabled constraint logic programs (see Section 4). Similar to the finite-state case, our formulation is a direct encoding of the definition of the bisimulation relations themselves. We describe how the programs can be evaluated using a constraint meta-interpreter implemented in XSB. Our experimental results show that symbolic bisimulation is practical for realistic systems. Surprisingly, our results show how even for relatively small finite-state systems, it may be better to perform symbolic bisimulation on its infinite-state counterparts. We conclude in Section 5 with a short discussion of the implications of this work.

2 Bisimilarity of Finite-State Systems

Labeled transition systems (LTSs) are widely used to capture the operational behavior of concurrent systems. An LTS is denoted by $L = (S, Act, \longrightarrow)$, where S is a finite set of states, Act is a finite set of *actions* (transition labels), and $\longrightarrow \subseteq S \times Act \times S$ is a transition relation. Transition from

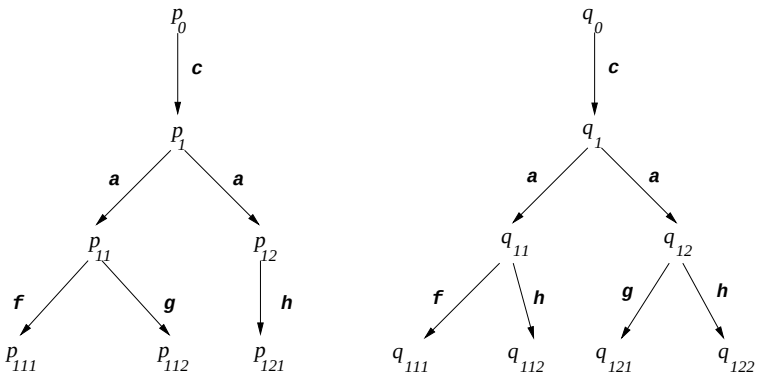

Figure 1: Example non-symbolic LTS

state s to t on an action a is represented by $s \xrightarrow{a} t$. Example LTSs are given in Figure 1.

An LTS $L = (S, Act, \longrightarrow)$ is encoded as a set of facts in a logic program P such that whenever $s \xrightarrow{a} t$, then `trans(`s`,` a`,` t`)` is in P. Note that since $s, t \in S$ as well as $a \in Act$ are from a finite set, they can be represented in a logic program by ground terms.

We first begin with a brief overview of bisimilarity in LTSs. Actions on transitions in LTS are of two types: actions that may be effected by external entity, environment, are called *observable actions* and actions that are the result of synchronization of subsystems are called *internal actions* or τ *actions*. Based on this notion of observability there are two variations of bisimilarity, *strong* and *weak*, described below.

2.1 Strong Bisimulation

Strong bisimulation does not differentiate between internal and observable actions.

Definition 1 (Bisimilarity Relation) *Given an LTS $L = (S, Act, \longrightarrow)$, $\mathcal{R}$ is a bisimilarity relation over L if*

$$\forall s_1, s_2 \in S.\ s_1 \mathcal{R} s_2 \Rightarrow (\forall (s_1 \xrightarrow{a} t_1).\ (\exists (s_2 \xrightarrow{a} t_2).\ t_1 \mathcal{R} t_2) \wedge s_2 \mathcal{R} s_1)$$

Two states in a system are equivalent with respect to bisimulation if they are related by *largest* bisimilarity relation $\mathcal{R}$. Two LTSs can be compared for bisimilarity by computing bisimulation of their disjoint union. For instance, consider the LTSs in Figure 1. States p_{11} and q_{11} are not bisimilar as there exists a transition from p_{11} with action g for which there is no matching transition from q_{11}. As such, states p_1 and q_1 are not bisimilar and also the states p_0 and q_0 are not bisimilar.

2.1.1 Encoding Strong Bisimulation

Using the dual of Definition 1, we can say that two states in a system are not equivalent with respect to bisimulation if they are related by the least relation $\overline{\mathcal{R}}$ defined as follows:

$$\forall s_1, s_2 \in S.\ s_1 \overline{\mathcal{R}} s_2 \Leftarrow (\exists (s_1 \xrightarrow{a} t_1).\ (\forall (s_2 \xrightarrow{a} t_2)\ \Rightarrow t_1 \overline{\mathcal{R}} t_2) \vee s_2 \overline{\mathcal{R}} s_1) \quad (1)$$

Note that $\overline{\mathcal{R}}$ can be encoded as a logic program by exploiting the least model computation of logic program as follows:

```
bisim(S1, S2)  :- tnot(nbisim(S1, S2)).

nbisim(S1, S2) :- trans(S1, A, T1),
                  no_matching_trans(S2, A, T1).
nbisim(S1, S2) :- nbisim(S2, S1).
```

In the above encoding, `nbisim/2` stands for $\overline{\mathcal{R}}$ defined by Equation 1. The goal `no_matching_trans(S2, A, T1)` stands for $\forall(s_2 \xrightarrow{a} t_2) \Rightarrow t_1 \overline{\mathcal{R}} t_2$ and is in turn defined as:

```
no_matching_trans(S2, A, T1) :-
     forall(trans(S2, A, T2), nbisim(T1, T2)). % T1 is not bisimilar to any T2
```

Note that since the terms `S2`, `A` are ground and `T2` is free, `forall/2` can be encoded without considering any free variables as follows:

```
forall(P, Q) :- findall(Q, P, L), all(L).

all([]).
all([Q|Qs]) :- Q, all(Qs).
```

2.2 Local Bisimulation

Evaluating `bisim(`s_1`,` s_2`)` using tabled resolution, we can prove or disprove bisimilarity of states s_1 and s_2. Note that goal directed computation with tabling makes the bisimulation checker "local": state space exploration is done only until the proof for bisimilarity or non-bisimilarity is obtained. However, if the given states are actually bisimilar, then we explore all the states reachable from s_1 and s_2. Another important aspect of our encoding is that it can be directly extended to symbolic bisimulation checking for infinite-state systems.

Given any two states in an LTS $(S, Act, \longrightarrow)$, the worst case time complexity of our bisimulation checker is $O(|S| \times |\longrightarrow|)$ assuming unit-time table lookups. The quadratic factor in our encoding comes from checking for bisimulation between (potentially) every pair of states. Table lookups may add $|S|^2$ factor to the complexity if tables are organized as a list, or $|\log(S)|$ factor if binary tree data structures are used. It should be noted that there are faster bisimulation checking algorithms: the Kanellakis-Smolka algorithm [5] runs in $O(|S| \times |\longrightarrow|)$; Paige and Tarjan's algorithm [9], implemented in CWB-NC, runs in $O(|\longrightarrow| \times \log|S|)$. These algorithms, unlike our implementation, compute equivalence classes of bisimilar states bottom up and are thus global.

2.3 Weak Bisimulation

Recall that, strong bisimulation does not differentiate between observable and internal actions. However, in practical settings two systems are considered to be equivalent when they are identical with respect to the observable actions. *Weak bisimulation* or *observational equivalence* formalizes this notion. It is defined on the basis of weak transition relation, which, in turn, is defined as follows:

Definition 2 (Weak Transition Relation) *Given a LTS $L = (S, Act, \longrightarrow)$, weak transition relation, $\longrightarrow_w \subseteq S \times Act \times S$, is the smallest relation such that*

1. $a \neq \tau$ *and* $s_1(\xrightarrow{\tau})^* s_2 \xrightarrow{a} s_3(\xrightarrow{\tau})^* t_1 \Rightarrow s_1 \xrightarrow{a}_w t_1$
2. $s_1(\xrightarrow{\tau})^* t_1 \Rightarrow s_1 \xrightarrow{\tau}_w t_1$

Note that (2) implies that for every state s, $s \xrightarrow{\tau}_w s$.

Definition 3 (Weak Bisimilarity) *Given an LTS $L = (S, Act, \longrightarrow)$, $\mathcal{R}_W$ is a weak bisimilarity relation over L, if*

$$\forall s_1, s_2 \in S.\ s_1 \mathcal{R}_W s_2 \Rightarrow (\forall (s_1 \xrightarrow{a} t_1).\ (\exists (s_2 \xrightarrow{a}_w t_2).\ t_1 \mathcal{R}_W t_2) \wedge s_2 \mathcal{R}_W s_1)$$

2.3.1 Encoding Weak Bisimulation

We begin with the encoding the weak transition relation. Note that $(\xrightarrow{\tau})^*$ is the transitive closure of $\xrightarrow{\tau}$ and can be encoded as follows:

```
taustar(S1, S1).
taustar(S1, S2) :-  taustar(S1, T), trans(T, tau, S2).
```

Using `taustar/2` weak transition relation can be directly encoded as follows:

```
weak_trans(S1, tau, T1) :- taustar(S1, T1).
weak_trans(S1, A, T1)   :- taustar(S1, S2),
                           trans(S2, A, S3),
                           A \= tau,
                           taustar(S3, T1).
```

Note that the only difference between Definitions 3 and 1 lies in the selection of matching transition, *i.e.*, $\exists (s_2 \xrightarrow{a}_w t_2).t_1 \mathcal{R}_W t_2$. Definition 1 uses strong transition $\xrightarrow{a}$, whereas Definition 3 uses weak transition $\xrightarrow{a}_w$. Thus the previous encoding of strong bisimilarity can be changed as follows to compute weak bisimilarity:

```
weak_bisim(S1, S2)  :- tnot(nweak_bisim(S1, S2)).

nweak_bisim(S1, S2) :- trans(S1, A, T1),
                       no_matching_trans(S2, A, T1).
nweak_bisim(S1, S2) :- nweak_bisim(S2, S1).

no_matching_trans(S2, A, T1) :-
                forall(weak_trans(S2, A, T2), nweak_bisim(T1, T2)).
```

2.4 Experimental Results

Below, we compare the performance of our local bisimulation checker with the one based on partition refinement algorithm [9] implemented in CWB-NC. Example systems selected are families of $stack(b, d)$ and $queue(b, d)$ with varying buffer lengths b and data domain sizes d. All measurements were made on a Sun 4U sparc Ultra Enterprise with 2G memory running Solaris 5.2.6, using XSB v2.3 and CWB-NC v1.11.

A $stack(b, d)$ is defined with a fixed buffer of fixed size b, where insert and delete actions respectively add and delete data to and from the top of the buffer. Whereas, in case of $queue(b, d)$ insert action adds data to the bottom of the buffer and delete action removes data from the top. Domain of each element in the buffer ranges over d distinct values. Figure 2 shows the transition system of a $stack(b, d)$ with $b = 2$ and $d = 2$, where $insert?1$ and $delete!1$ represents input and output actions that insert and delete data value 1 to and from the stack.

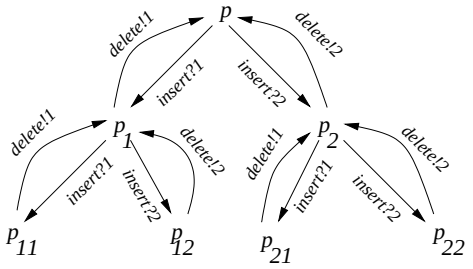

Figure 2: $stack(2, 2)$

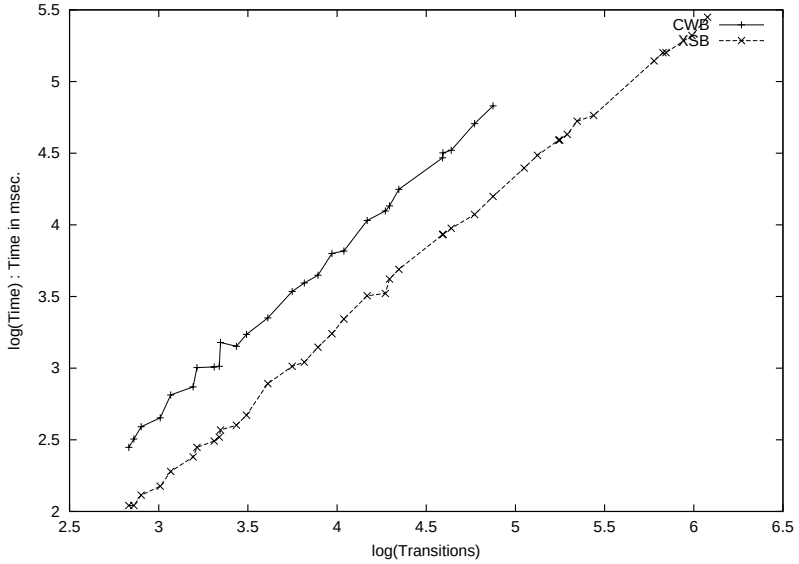

Figure 3: Bisimulation of two identical $stack(b, d)$s in XSB and CWB-NC

Figure 3 shows the time taken to check for bisimilarity of two identical $stack(b, d)$s for different combinations of b and d. In the figure, the log of time is plotted against the log of number of transitions. The number of transitions in the system is $O(d^b)$. Since the systems are bisimilar, both our encoding and the CWB-NC explore the entire state space. As shown in Figure 3, XSB implementation is roughly 3 times faster than CWB-NC implementation and hence comparable.

We now present the time taken to check bisimilarity of $stack(b, d)$ and $queue(b, d)$ for different combinations of b and d. These systems are not bisimilar when $b \geq 2$ and $d \geq 2$. In this case local bisimulation checker implemented in XSB outperforms global checking algorithm implemented in CWB-NC. The XSB implementation can check for bisimilarity of $stack(b, d)$ and $queue(b, d)$ for $b = 5000$, $d = 5$ (Table 1) and with CWB-NC the largest system we can check is of $b = 7$, $d = 4$ (Table 2). It is worth mentioning here that local bisimulation checking in case of $stack(b, d)$ and $queue(b, d)$ is independent of d. An inspection of query evaluation in XSB reveals that only two elements from the data domain d are considered even when $d \geq 2$.

XSB	Buffer Length (b)				
Data Domain (d)	1000	2000	3000	4000	5000
2	1.68	6.20	13.08	23.69	40.97
3	1.71	6.45	13.27	23.84	41.40
4	1.80	6.41	13.85	23.90	40.49
5	1.66	6.47	13.27	24.25	41.70

Table 1: Strong bisimulation of $stack(b, d)$ and $queue(b, d)$ in XSB (table entries are time in secs.)

CWB-NC	Buffer Length (b)					
Data Domain (d)	3	4	5	6	7	8
2						0.57
3		0.14	0.42	1.37	4.60	15.87
4	0.10	0.42	1.87	8.28	40.50	-
5	0.21	1.19	6.26	-		
6	0.38	2.61	-			
7	0.66	5.26	-			
8	1.09	9.90	-			
9	1.73	-				
10	2.58	-				

Table 2: Strong bisimulation of $stack(b, d)$ and $queue(b, d)$ in CWB-NC (table entries are time in secs.)

3 Infinite-State Systems

Traditionally model checking and bisimulation algorithms are formulated on the basis of finite LTSs. However, an LTS for even a relatively small concurrent system may be very large. We introduce the notion of *Symbolic Transition Systems* (STSs) as a way to represent large or infinite-state systems. An STS can be viewed as an LTS augmented with state variables, guards on transitions, and nonground terms as action labels. Infinite-state systems can be represented finitely by STSs.

3.1 Symbolic Transition Systems

We assume the standard notion of terms, substitutions and unifiers. We use $\mathcal{V}$ to denote an enumerable set of variables, $\mathcal{F}$ to denote a set of function symbols, $\mathcal{P}$ to denote a set of predicate symbols, and $\mathcal{B}$ to denote $\{true, false\}$. Function and predicate symbols have fixed arity; function symbols of arity 0 are called constants. *Expressions*, denoted by $\mathcal{E}$ are terms over $\mathcal{F} \cup \mathcal{V}$ and *guards*, denoted by γ, are terms over $\mathcal{P} \cup \mathcal{F} \cup \mathcal{V}$ where predicate symbols appear at (and only at) the root. The set of variables in a term t is denoted by $vars(t)$. Substitutions, denoted by θ and σ (possibly primed or subscripted), are mappings from $\mathcal{V}$ to $\mathcal{E}$. A substitution that maps value v to variable x is written as $[v/x]$. A term t under substitution σ is denoted by $t\sigma$; the composition of two substitutions σ_1, σ_2 is denoted simply by $\sigma_1\sigma_2$.

A guard γ of arity n is interpreted as a mapping from $\mathcal{E}^n$ to $\mathcal{B}$. Alternatively, γ can be viewed as a set of substitutions such that $\sigma \in \gamma$ iff $\gamma\sigma = true$. An *action* is a term in one of the following forms:

$$
\begin{array}{rcl}
s_0 & \xrightarrow{insert?x} & s_1(x) \\
s_1(x) & \xrightarrow{delete!x} & s_0 \\
s_1(x) & \xrightarrow{insert?y} & s_2(x,y) \\
s_2(x,y) & \xrightarrow{delete!y} & s_1(x)
\end{array}
\qquad
\begin{array}{rcl}
q_0 & \xrightarrow{insert?x} & q_1(x) \\
q_1(x) & \xrightarrow{delete!x} & qs_0 \\
q_1(x) & \xrightarrow{insert?y} & q_2(x,y) \\
q_2(x,y) & \xrightarrow{delete!x} & q_1(y)
\end{array}
$$

Figure 4: Example STS representing 2-length stack (a) and 2-length queue (b) over arbitrary data domain

- *Input Action:* Represented as $c?x$, where c is a constant and x is a variable.
- *Output Action:* Represented as $c!e$, where c is a constant and e is an expression.
- *Internal Action:* Represented by τ, a constant.

Output actions without the expression parameter, as in $c!$, are also known signals, and are simply represented as c.

Definition 4 (Symbolic Transition System) *A symbolic transition system is a finite labeled directed graph $(S, \longrightarrow)$, where S is a set of terms, called* locations*, which form vertices of the graph, and $\longrightarrow$ is the edge relation where each edge $s \xrightarrow{\gamma,\alpha,\rho} t$ is labeled with:*

- *an action α such that*
  - *$vars(\alpha) \subseteq vars(s)$ if α is not an input action.*
  - *$vars(\alpha) \cap vars(s) = \{\}$ if α is an input action.*
- *a guard γ such that $vars(\gamma) \subseteq vars(s)$, and*
- *a transfer relation ρ that relates $vars(s)$ to $vars(t)$.*

If a guard is *true* it is omitted. The transfer relation is used to model updates to variables. The transfer relation is omitted whenever it is the identity mapping over the source and target variables. The edge relation of two STSs representing a stack and a queue that stores arbitrary data values, with maximum buffer length of 2, are shown in Figure 4 (a) and (b) respectively.

Note that the definition of STSs is general enough to capture Symbolic Transition Graphs (STGs) [4] and STGs with Assignments (STGAs) [7]. For instance, STGs are STSs where each edge $s \xrightarrow{\gamma,\alpha,\rho} t$ is such that $vars(t) \subseteq (vars(s) \cup vars(\alpha))$.

3.2 Semantics of STS

Semantics of an STS $\mathcal{S}$ is given in terms of a transition relation, denoted by $\mathcal{T}(\mathcal{S})$, which is generated by interpreting $\mathcal{S}$ with respect to substitutions. Given an STS $\mathcal{S}$, each state in $\mathcal{T}(\mathcal{S})$ is a location s paired with a substitution σ on $vars(s)$. There are different variants of semantics depending on how variables are interpreted. In the following, we describe *late* and *early* semantics which are the most widely studied to date.

Late semantics is a natural interpretation of the symbolic transition systems, by "reading off" transitions from a state $s\sigma$ by applying the substitution on all components of edges of the form $s \xrightarrow{\gamma,\alpha,\rho} t$ from location s. This is captured formally by the following definition.

Definition 5 (Late Transition Relation) *Let σ be a substitution such that $s \xrightarrow{\gamma,\alpha,\rho} t \in \mathcal{S}$, $\sigma \Rightarrow \gamma$, and σ satisfies ρ. Then $\mathcal{T}(\mathcal{S})$ contains $s\sigma \xrightarrow{\alpha\sigma}_l t\sigma$.*

One interesting aspect of late semantics is that we only capture substitutions on variables in the target state of a transition if they are related by ρ to those in the start state. For instance, consider an input transition of the form $s \xrightarrow{c?x} t$. From definition of STS, $x \notin vars(s)$. If t contains x, then x does not immediately pick up a value due to this transition. The variable x is left to be bound by a guard or transfer relation in a subsequent state. *Early* semantics interprets the new variables introduced on input actions by immediately assigning values to them.

Definition 6 (Early Transition Relation) *Let σ be a substitution such that $s \xrightarrow{\gamma,\alpha,\rho} t \in \mathcal{S}$, $\sigma \Rightarrow \gamma$, and σ satisfies ρ. Then $\mathcal{T}(\mathcal{S})$ contains*

$$\begin{array}{ll} s\sigma \xrightarrow{\alpha\sigma}_e t\sigma & \text{if } \alpha \text{ is not an input action} \\ s\sigma \xrightarrow{c?v}_e t\sigma[v/x] & \text{if } \alpha = c?x, \text{ and } v \text{ is a ground term} \end{array}$$

The two semantics naturally yield two variants of the bisimulation relation, as described in Section 4. Below, we describe how an STS can be encoded as a logic program so that the late semantics can be computed directly by resolution.

Encoding STS as a Constraint Logic Program: The edge relation of an STS $\mathcal{S}$ can be encoded as a constraint logic program P such that for each $s \xrightarrow{\gamma,\alpha,\rho} t \in \mathcal{S}$

```
sts_edge(s, α, γ, ρ, t)
```

is a fact in P. We can encode each guard γ as a predicate in P so that whether $\sigma \Rightarrow \gamma$ can be checked using the query $\gamma\sigma$. We can also encode the transfer relation ρ as a predicate in P.

The late transition relation (Definition 5) can be computed from this set of facts using the following rule:

```
late_trans(S, A, T) :-
     sts_edge(S, A, Gamma, Rho, T),
     Gamma,   % The guard is satisfied
     Rho.     % and so is the transfer relation
```

Early transition relation cannot be so directly encoded due to the universal quantifier over values in its definition (see Definition 6).

In our encoding of the bisimulation relations over STSs, it becomes necessary to explicitly obtain conditions under which a late transition is enabled. We call this as a *symbolic transition*, denoted by `strans` defined by the following rule:

```
strans(S, A, Gamma, T) :-
     sts_edge(S, A, Gamma, Rho, T),
     Rho.     % The transfer relation is satisfied
```

The use of `strans` in the encoding of bisimulation relations is described in Section 4. In fact the bisimulation relation uses only `strans` and `late_trans` and do not directly operate on STSs (i.e. use `sts_edge`).

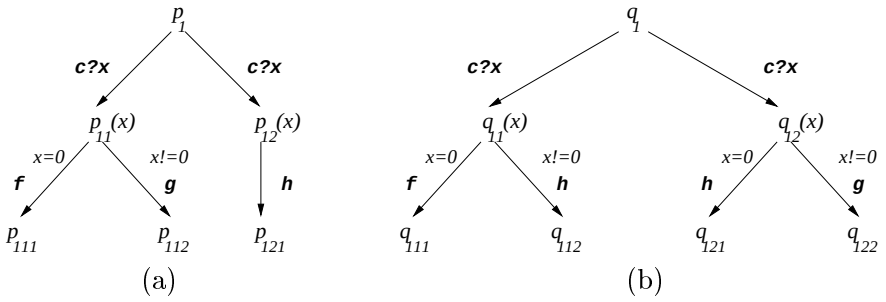

Figure 5: Example symbolic transition systems

4 Symbolic Bisimulation

Late bisimulation and early bisimulation, which we describe in detail below, differ in the way input actions are treated. Consider checking the bisimilarity of locations p_1 and q_1 in the STSs given in Figure 5. Clearly, locations $p_{111}, p_{112}, p_{121}$ are all bisimilar to locations $q_{111}, q_{112}, q_{121}, q_{122}$ (all are deadlocked). Furthermore, location p_{11} is bisimilar to q_{11} when $x = 0$, and is bisimilar to q_{12} if $x! = 0$; location p_{12} is bisimilar to q_{11} when $x! = 0$; and is bisimilar to q_{12} when $x = 0$. However, are p_1 and q_1 bisimilar?

When q_1 makes a transition, say $q_1 \xrightarrow{c?x} q_{11}$, what is the matching transition from p_1? According to the bisimilarity sets we have computed so far, the matching transition is the one to p_{11} when $x = 0$ and the one to p_{12} when $x! = 0$. These two transitions together cover the transition from q_1 to q_{11}. However, note that the action on this transition is an input: $c?x$. Do we know enough about the value of x to make the choice between p_{11} and p_{12}? According to early semantics, the value of x is known when a transition is taken. However, according to late semantics, the value of x is determined only by later guards, and hence unknown when the transition is taken. Hence, p_1 and q_1 are *early-bisimilar* but not *late-bisimilar*.

Before a formal presentation of the bisimulation relations over STSs, we motivate their definitions by starting from the basic bisimulation relation for the finite-state case. In Definition 1, we had

$$\forall s_1, s_2 \in S \; s_1 \mathcal{R} s_2 \Rightarrow (\forall s_1 \xrightarrow{a} t_1 \; (\boxed{\exists s_2 \xrightarrow{a} t_2} \; t_1 \mathcal{R} t_2) \wedge s_2 \mathcal{R} s_1)$$

The question we have now is, having picked a transition $s_1 \xrightarrow{a} t_1$, how do we pick the matching $s_2 \xrightarrow{a} t_2$; and since in the symbolic case the action labels may bind variables, under what substitution. In the late bisimulation case, recall that the variable in an input label is bound only afterward, and hence the matching transition $s_2 \xrightarrow{a} t_2$ should be such that t_1 and t_2 are bisimilar under all substitutions to the input variable. In summary, the matching transition must be picked before considering substitutions. This intuition is captured by the following formal definition of late bisimulation.

Definition 7 (Late Bisimulation) *Given an STS $(S, \longrightarrow)$, the late bisimulation relation with respect to substitution θ, denoted by $\mathcal{R}_l^\theta$, is a subset of $S \times S$ such that*

$$s_1 \mathcal{R}_l^\theta s_2 \quad \Rightarrow \quad (\forall s_1\theta \xrightarrow{\alpha_1}_l t_1\theta \quad \boxed{\exists s_2\theta \xrightarrow{\alpha_2}_l t_2\theta} \; \boxed{\forall \sigma} \; (\alpha_1[\theta\sigma] = \alpha_2[\theta\sigma]) \wedge t_1 \mathcal{R}_l^{\theta\sigma} t_2) \wedge \; s_2 \mathcal{R}_l^\theta s_1$$

In the early bisimulation case, recall that the variable in an input action is bound at the transition itself, there is no choice to make in terms of substitutions. This is captured by the following formal definition of early bisimulation.

Definition 8 (Early Bisimulation (using $\longrightarrow_e$)) *Given an STS $(S, \longrightarrow)$, the early bisimulation relation with respect to substitution θ, denoted by $\mathcal{R}_e^\theta$, is a subset of $S \times S$ such that*

$$s_1 \mathcal{R}_e^\theta s_2 \quad \Rightarrow \quad (\forall s_1\theta \xrightarrow{\alpha_1}_e t_1\theta \; \boxed{\exists s_2\theta \xrightarrow{\boxed{\alpha_1}}_e t_2\theta} \; (t_1 \mathcal{R}_e^\theta t_2)) \wedge \; s_2 \mathcal{R}_e^\theta s_1$$

The above definition relies on the definition of the early transition relation. It turns out, however, that we can define early bisimulation completely in terms of the late transition relation [10], as follows:

Definition 9 (Early Bisimulation (using $\longrightarrow_l$)) *Given an STS $(S, \longrightarrow)$, the early bisimulation relation with respect to substitution θ, denoted by $\mathcal{R}_e^\theta$, is a subset of $S \times S$ such that*

$$s_1 \mathcal{R}_e^\theta s_2 \quad \Rightarrow \quad (\forall s_1\theta \xrightarrow{\alpha_1}_l t_1\theta \quad \boxed{\forall\sigma} \; \boxed{\exists s_2\theta \xrightarrow{\alpha_2}_l t_2\theta} \; (\alpha_1[\theta\sigma] = \alpha_2[\theta\sigma]) \wedge t_1 \mathcal{R}_e^{\theta\sigma} t_2) \wedge \; s_2 \mathcal{R}_e^\theta s_1$$

This alternative definition of early bisimulation is especially important to a logic-programming-based encoding since early transition relations are hard to encode as logic programs.

4.1 Encoding Bisimulation Checkers as Logic Programs

We can encode checkers for equivalence with respect to late as well as early bisimulation following the encoding of bisimulation checkers for the finite-state case.

Early Bisimulation: Consider the complement of early bisimulation relation $\mathcal{R}_e$, written as $\overline{\mathcal{R}_e}$:

$$s_1 \overline{\mathcal{R}_e^\theta} s_2 \; \Leftarrow \; (\exists s_1\theta \xrightarrow{\alpha_1}_l t_1\theta \quad \exists\sigma \; \forall s_2\theta \xrightarrow{\alpha_2}_l t_2\theta \; (\alpha_1[\theta\sigma] = \alpha_2[\theta\sigma]) \Rightarrow \; t_1 \overline{\mathcal{R}_e^{\theta\sigma}} t_2) \vee \; s_2 \overline{\mathcal{R}_e^\theta} s_1 \qquad (2)$$

Since bisimulation is the largest such relation, the complement is naturally the least relation that satisfies the above equation. This relation can be encoded as a constraint logic program as follows:

```
nbisim(S1, S2) :-
       late_trans(S1, A1, T1),
       no_matching_trans(S1, A1, T1, S2).
nbisim(S1, S2) :-
       nbisim(S2, S1).

no_matching_trans(S1, A1, T1, S2) :-
       forall((A2, T2),
                late_trans(S2, A2, T2),
                nsimulate(A1, T1, A2, T2)).

nsimulate(A1, T1, A2, T2) :-
       similar_act(A1,A2), nbisim(T1, T2)
     ; not_similar_act(A1,A2).
```

Several differences are apparent between the finite-state bisimulation checker in Section 2 and the one given above. The first and most obvious difference is the use of `late_trans` for `trans`. The second is the use of a ternary `forall` predicate in order to explicitly differentiate between the bound and free variables. Note that in the finite-state case, there were no free variables in the universally quantified formula, and hence we could vastly simplify the way `forall` was encoded. In the infinite-state case we need to find consistent values for all the free variables used in the universally quantified formula ($\forall s_2\theta \xrightarrow{\alpha_2}_l t_2\theta \ldots$ in Equation 2). The more complicated encoding of `forall` to accommodate free variables as well as to process constraints is discussed in Section 4.2.

The third difference is the use of `similar_act` (and `not_similar_act`) to check for $(\alpha_1[\theta\sigma] = \alpha_2[\theta\sigma])$ in Equation 2 (and its negation). Although `similar_act(A1,A2)` is `A1=A2`, `not_similar_act(A1,A2)` is not simply the negation of `A1=A2`, for the following reason. Two output actions $c!x$ and $c!y$ can be dissimilar as long as x and y can be bound to different values. Note that, in contrast, since an input action creates a new bound variable, $c?x$ and $c?y$ are always similar. Hence, we have the following encoding for `similar_act` and `not_similar_act`:

```
similar_act(A1, A2) :- A1 = A2.

not_similar_act(A1, A2) :-
        A1 \= A2,
        (   (A1 = in(C,_), (A2 = in(D,_), C\=D
                        ; A2 = out(_,_)
                        ; A2 = tau)
           ; A1 = out(_,_)
           ; A1 = tau)).
```

Late bisimulation: Let us now consider $\overline{\mathcal{R}_l}$, the complement of late bisimilarity relation:

$$s_1\overline{\mathcal{R}_l^\theta}s_2 \Leftarrow (\exists s_1\theta \xrightarrow{\alpha_1}_l t_1\theta \quad \forall s_2\theta \xrightarrow{\alpha_2}_l t_2\theta \ \exists\sigma \ (\alpha_1[\theta\sigma] = \alpha_2[\theta\sigma]) \Rightarrow t_1\mathcal{R}_l^{\theta\sigma}t_2) \vee \ s_2\overline{\mathcal{R}_l^\theta}s_1 \qquad (3)$$

The essence of this equation is that in order to show non-bisimilarity, for every transition from s_2 to t_2 we should find a *local* σ such that either the actions do not match, or t_1 and t_2 are non-bisimilar. This condition can be tested by simply ensuring that the different transitions from s_2 are standardized apart before checking for matching contexts. Standardization can be done via `copy_term/2` which generates a copy of a term with fresh variables. Late bisimulation can thus be derived from the encoding of early bisimulation by modifying `nsimulate/5` as follows:

```
nsimulate(A1, T1, A2, T2) :-
      ( similar_act(A1,A2),
        change_environments(A1, (T1,T2), (U1,U2)),
        nbisim(U1, U2))
    ; not_similar_act(A1,A2).

change_environments(in(_,_), E1, E2) :- copy_term(E1, E2).
change_environments(out(_,_), E1, E1).
change_environments(tau, E1, E1).
```

The predicate `change_environments/3` ensures that each transition on input action from s_2 is evaluated in a separate environment, as required by late bisimulation.

Discussion: Observe that the nested call to `nbisim/2` in the definition of `nsimulate` inherits a new set of constraints from the guards on the two selected transitions as well as the values under which the actions are similar. In our encoding, the current context in which `nbisim/2` is evaluated is maintained implicitly. This is a useful simplification as compared to the original algorithm of Hennessy and Lin [4], where the context of the bisimulation is maintained explicitly. The Hennessy-Lin algorithm returns the most general context under which the two processes are bisimilar. In a similar vein, when our encoding detects that two processes are not bisimilar, we can retrieve the context which witnesses the non-bisimilarity of the two processes.

The complexity of the evaluation is $O(|S| \times | \longrightarrow |)$ assuming unit-time table look up and constraint manipulation, which is same as Hennessy and Lin's procedural algorithm [4]. Furthermore, the encoding clearly separates the logical aspects of bisimulation from its representational aspects.

4.2 Implementation

The encoding can be directly executed in a tabled constraint logic programming system that implements `forall` faithfully. Note that even if the guards and transfer relation of an STS contain only equality constraints, the bisimulation checker itself needs to handle both equality and disequality constraints.

A Meta-Interpreter for Tabled Constraint Logic Programs: We have implemented a constraint meta-interpreter that handles tabled logic programs over equality and disequality constraints in XSB. The meta-interpreter maintains the constraint store and simplifies the constraints as they are propagated, thus simulating a tabled CLP environment. We use the traditional trick of trading off the costs associated with maintaining constraint stores always in canonical form against the cost of extra resolution steps due to undetected inconsistencies in non-canonical constraint stores. The distinctive feature of the meta-interpreter is the encoding of the `forall` construct. The basis of this construct is the following implementation of `forall` that correctly treats free variables in the quantified formula over equality domain:

```
forall(BoundVars, Antecedent, Consequent) :-
        bagof(BoundVars, Antecedent, BindingList),
        excess_vars(Consequent, BoundVars, [], FreeVars),
        all_true(BindingList, BoundVars, FreeVars, Consequent).
all_true([Binding|Rest], BoundVars, FreeVars, Consequent) :-
        copy_term(f(Consequent, BoundVars, FreeVars),
                        f(Copy, Binding, FreeVars)),
        Copy,
        all_true(Rest, BoundVars, FreeVars, Consequent).
all_true([], _, _, _).
```

The standard Prolog predicate `bagof` is used to collect the set of bindings on the bound variables for each valuation of the free variables such that `Antecedent` is true. Predicate `excess_vars` collects the variables in the first argument that do not occur in the second, and is used to simply find the set of free variables in a term, given the set of bound variables. For each binding in the set obtained in `bagof`, `Consequent` is evaluated under the proper substitution of bound variables. The `copy_term` is used to separate the environments for evaluating `Consequent` for the different bindings on the bound variables. Note that the free variables are shared between the evaluations, thereby ensuring that `forall` is evaluated for consistent valuations of the free variables. This encoding of `forall` is lifted to the constraint meta-interpreter by adding constraint stores and interpretation of `bagof` and `copy_term` over constraint stores. Details are omitted.

Changes to the encoding to use the meta-interpreter: Note that the `forall` construct is interpreted using an all-solutions predicate (`bagof` in this case). If we use this implementation of `forall` in order to evaluate `no_matching_trans/4`, note that we will collect constraints under which there are no matching transitions only considering those transitions that are enabled from `S2`. Hence, we need to separate enabling and disabling of transitions from the actual computation of substitutions. We do so by using, instead of `late_trans/3`, the predicate `strans/4` which explicitly returns the enabling condition of a transition without evaluating it. For instance, the implementation of early bisimulation is changed to

```
nbisim(S1, S2) :-
       strans(S1, A1, Gamma1, T1), Gamma1,
       no_matching_trans(S1, A1, T1, S2).
nbisim(S1, S2) :-
       nbisim(S2, S1).

no_matching_trans(S1, A1, T1, S2) :-
       forall((A2, Gamma2, T2),
                strans(S2, A2, Gamma2, T2),
                nsimulate(A1, T1, A2, Gamma2, T2)).

nsimulate(A1, T1, A2, Gamma2, T2) :-
          (Gamma2,     (similar_act(A1,A2), nbisim(T1, T2))
                    ;  not_similar_act(A1,A2))
       ;  negate(Gamma2).
```

where `negate(`p`)` evaluates the constraints under which p fails. The encoding of late bisimulation is also modified similarly.

4.3 Experimental Results

We measured the performance of our symbolic bisimulation checker on an infinite-state version of stack and queue. Stacks and queues, denoted by $stack(n)$ and $queue(n)$, of different buffer lengths (n) but with unspecified domain, were defined as STSs (e.g., see Figure 4) and encoded as `sts_edge` facts. Note that even for fixed values of n, $stack(n)$ and $queue(n)$ are infinite-state systems since each element in them can store arbitrary data values. Tables 3 and 4 show the time performance for checking *early* bisimulation comparing $stack(n)$ and $queue(n)$, and $stack(n)$ and $stack(n)$ respectively. Times for *late* bisimilarity checking are about 2% more than their early bisimulation counterpart due to the extra `copy_term` overhead.

Buffer Length (n)							
10	20	30	40	50	60	70	80
0.42	2.15	6.28	14.22	27.99	49.69	80.38	124.42

Table 3: Time for symbolic bisimulation checking for of $stack(n)$ and $queue(n)$

At first sight, these tables display an anomaly: time taken to check bisimilarity when two systems are not bisimilar (Table 3) is more than that for systems which are bisimilar (Table 4) for the same buffer length. This appears to contradict our previous observation that local bisimulation explores less state space and takes less time when the systems under consideration are not bisimilar

Buffer Length (n)							
10	20	30	40	50	60	70	80
0.16	0.54	1.19	2.22	3.68	5.60	8.14	11.37

Table 4: Time for symbolic bisimulation checking for of $stack(n)$ and $stack(n)$

as compared to the case when the systems are bisimilar. However closer inspection reveals that the symbolic state space explored in checking for bisimilarity between two stacks is much less than symbolic state space explored when checking for bisimilarity between a stack and a queue. In fact, the symbolic (global) state space explored for checking bisimilarity between two stacks is linear in the buffer length. In contrast, the proof for non-bisimilarity of stack and queue depends both on buffer length and data domain size. It is worth mentioning that the state space explored for local bisimilarity checking depends greatly on the way the two transition systems are encoded. In case of checking for bisimilarity between a queue and a stack, if we first explore transitions with output (*delete*) actions before exploring those with input (*insert*) actions the states needed to be explored before the first non-similarity is detected is independent of the buffer length.

It should also be noted that the symbolic state space of these systems may, in fact, be smaller than the ground state space even for data domain sizes as low as 2. For instance, consider the symbolic state space of a 2-element queue in Figure 4(b). Its symbolic state space has 3 states, since the states $q_1(x)$ and $q_1(y)$ are simply variants of each other (i.e., identical modulo names of variables), and hence identified as a single symbolic state. In contrast, even for a 2-element data domain, say $\{1, 2\}$, observe that $q_1(1)$ is a state distinct from $q_1(2)$. Moreover, $q_2(x, y)$ and $q_2(y, x)$ represent the same symbolic state, while $q_2(1, 2)$ and $q_2(2, 1)$ behave differently. This is one of the key reasons why we can do symbolic bisimulation checking comparing two stacks with buffer length as high as 80 in around 11 seconds, whereas in the non-symbolic case, even comparing two stacks with buffer length of 18 each and with data domain size of just 2 explores over over $250K$ states, taking over 270 seconds in that process.

Finally, the meta-interpretation of constraints in the symbolic bisimulation imposes a heavy performance overhead. Using the symbolic bisimulation checker for LTSs (i.e., ground transition systems) is nearly 60 times slower than using the finite-state bisimulation checker. It is expected that a cleverer encoding of the constraint solver, together with the use of attributed variables [2] to integrate the solver with the LP engine, will significantly lower these overheads.

5 Conclusion

In this paper we demonstrated how the power and versatility of tabled logic programming can be used for checking bisimilarity of infinite-state systems in a natural way. Our implementation is goal-directed, i.e., we explore only states needed to prove or disprove the bisimilarity of the given states, and it can handle both early and late versions of strong as well as weak bisimilarity. Furthermore, the complexity of this implementation matches that of Hennessy and Lin's algorithm modulo table-lookup time. Our experimental results show that the symbolic bisimulation checker over an infinite-state system can be considerably more efficient to use compared to regular bisimulation checking over LTSs generated by finite instances of these systems, even for relatively small domain sizes. Applying the symbolic checker to real-life verification problems thus appears feasible despite significant overheads imposed by the constraint solver.

A recent paper [8] explored the use of constraint logic programs for checking bisimilarity of

timed systems, where timed systems are encoded by their corresponding transition relation. The encoding used in that work can be seen as specializing the `nbisim` predicate with respect to the transition relation. It is therefore expected that our encoding can be used for checking bisimilarity of timed systems. However, the performance of the checker will crucially hinge on the performance of a constraint solver for linear constraints needed to evaluate queries over timed systems.

References

[1] CADP. Caesar/Aldebaran Development Package c1.112, 2001. Available from `http://www.inrialpes.fr/vasy/cadp.html`.

[2] B. Cui and D.S. Warren. A system for tabled constraint logic programming. In *Computational Logic*, pages 478–492, 2000.

[3] CWB-NC. The Concurrency Workbench of New Century v1.1.1, 2001. Available from `www.cs.sunysb.edu/~cwb`.

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

[5] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, May 90.

[6] Z. Li and H. Chen. Computing strong/weak bisimulation equivalences and observation congruence for value-passing processes. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 300–314, 1999.

[7] H. Lin. Symbolic transition graphs with assignments. In *Concurrency Theory (CONCUR)*, pages 50–65, 1996.

[8] S. Mukhopadhyay and A. Podelski. Constraint database models characterizing timed bisimilarity. In *Practical Applications of Declarative Language*, 2001.

[9] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, December 1987.

[10] J. Parrow. An introduction to π-calculus. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 8, pages 479–544. North-Holland, 2001.

[11] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. W. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV '97)*, Haifa, Israel, July 1997. Springer-Verlag.

[12] C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, Y. Dong, X. Du, A. Roychoudhury, and V.N. Venkatakrishnan. XMC: A logic-programming-based verification toolset. In *Computer Aided Verification (CAV)*, 2000.

[13] XSB. The XSB logic programming system v2.3, 2001. Available from `http://www.cs.sunysb.edu/~sbprolog`.