Symbolic Bisimulation using Tabled Constraint Logic Programming

Madhavan Mukund, C.R. Ramakrishnan,
I.V. Ramakrishnan, Rakesh Verma

Abstract

Bisimulation equivalence is a standard way of comparing concurrent systems. Although the problem of checking bisimulation equivalence of finite-state systems is well-studied, the situation is not so clear for infinite-state systems. Systems with value-passing over infinite domains are inherently infinite-state. Hennessy and Lin have proposed a symbolic representation which often results in finite descriptions of value-passing systems. In this paper, we investigate the problem of checking bisimulation for such symbolic transition systems using tabled constraint logic programming.

Keywords: Bisimulation, value-passing calculi, infinite-state system, tabled evaluation, logic programming.

1 Introduction

A tabled logic programming system offers an attractive platform for encoding computational problems in the specification and verification of systems. Model checking, an automatic technique for verifying if a finite-state concurrent system specification satisfies a property expressed as a temporal logic formula, constitutes one such important class. Using the XSB tabled logic-programming system we developed XMC, a local model checker for a CCS-like value-passing system specification language and the modal mu-calculus temporal logic [8]. XMC, written in under 200 lines of tabled Prolog code, has been operational for over a year now. Our experience with using XMC for specification and verification of industrial-strength protocols strongly suggests that implementing practical model checkers using tabled logic programming is indeed feasible [9].

The first author is at Chennai Mathematical Institute, Chennai, India; E-mail: `madhavan@smi.ernet.in`. The second and third authors are at Dept. of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794-4400; E-mail: {`cram,ram`}`@cs.sunysb.edu`. The fourth author is at Department of Computer Science, University of Houston, TX 77204. E-mail: `rmverma@cs.uh.edu`.

The first author was partially supported by NSF Grant CDA-9805735. The second and third authors were partially supported by NSF Grants CCR-9711386 and EIA-9705998. The second author was also supported by NSF Career Grant CCR-9876242. The fourth author was partially supported by NSF grant CCR 9732186.

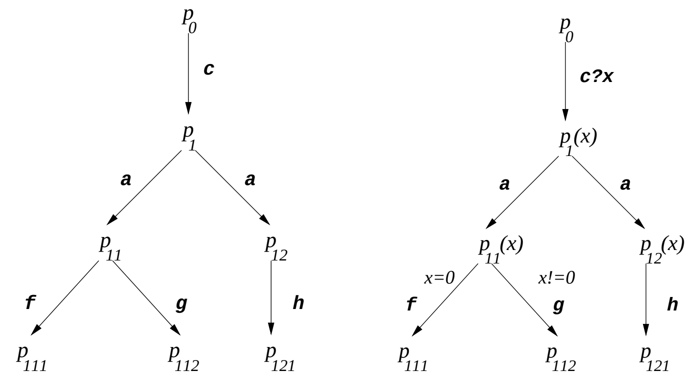

Figure 1: Examples of (a) non-value-passing and (b) symbolicd LTSs

Having established the applicability of tabled LP for model checking we have now begun investigating its applicability for checking *bisimulation equivalence* [6] which is another problem of fundamental importance in verification. Informally, given a pair M, M' of automata we say that they are bisimilar if for every transition in M there exists a corresponding transition in M' and vice versa. Consider the class of logics expressible in the mu-calculus—all standard modal and temporal logics used in verification fall into this class. Given a logic L in this class and a structure M, if M' is bisimilar to M then M' satisfies exactly the same set of formulas of the logic L as M. Bisimilar systems play a very useful role in coping with the state explosion that arises in model checking since one can use a much smaller system M' instead of M for verifying system properties.

There has been a lot of research on efficient algorithms for bisimulation testing. 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 many practical problems that arise in verification are often infinite-state where both M and M' are no longer finite-state systems. Hennessy and Lin were the first to consider the problem of bisimilarity testing of infinite-state systems [2] in the setting of value-passing languages and Lin has recently expanded their initial work further [5, 4]. Nevertheless research on this problem remains in a state of infancy.

In this extended abstract we explore the use of logic programming for the above problem. We show how we can use the power and versatility of tabled logic programming augmented with constraints for testing bisimilarity of infinite-state systems. Our approach handles all four possible definitions of bisimilarity for value-passing systems obtained by qualifying strong and weak bisimilarity with the notions early/late.

2 Symbolic Bisimulation

Labeled transition systems (LTSs) are widely used to capture the operational behavior of concurrent systems. An LTS over a set of states S and actions A is a finite directed graph, with node and edge labels. In the non-value-passing case (e.g., basic CCS [6]), nodes are labeled with symbols from S and the edges are labeled with symbols from A. An LTS is usually denoted by $L = (S, T)$ where S is the set of nodes (i.e., states) and $T \subseteq S \times A \times S$ is the edge relation (with labels). An example LTS is given in Figure 1(a).

In the value-passing case, the LTSs are *symbolic*: a node represents a set of states, and an edge represents transitions between various subsets of the states denoted by the source and destination nodes. The node labels are, in general, terms with free variables. Edge labels are guarded actions such that all free variables in the guard and action occur free in the source node, and all free variables in the destination node occur as free variables in the source node, or are bound by the action. An example of a symbolic LTS (from [2]) is shown in Figure 1(b). In the figure, we can go from state p_0 to $p_1(x)$ after performing an input action $c?x$. The input action binds x. From state $p_{11}(x)$ we can go to p_{111} after doing a f action, provided $x = 0$ (the guard on the transition). Guards that are always true are usually omitted.

2.1 Bisimilarity in non-value-passing systems

We first begin with a brief overview of similarity and bisimilarity in non-value-passing LTSs. We define these notions in a single LTS; two LTSs can be compared by computing the similarity relations on their disjoint union.

Given an LTS $L = (S, T)$, $\mathcal{R}$ is a similarity relation over L, denoted by $sim(\mathcal{R})$ iff

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

$\mathcal{R}$ is a bisimilarity relation (denoted by $bisim(\mathcal{R})$) iff $\mathcal{R}$ is a similarity relation and is symmetric, i.e., $sim(\mathcal{R})$ and $\forall s_1, s_2.\ (s_1 \mathcal{R} s_2 \Leftrightarrow s_2 \mathcal{R} s_1)$. Among the many bisimilarity relations, the largest one is of most interest. Using the dual of the definition of sim, we can define the largest bisimilarity relation as follows.

Definition 1 (Largest Bisimilarity Relation) *Given an LTS $L = (S, T)$, $\mathcal{R}$ is the largest bisimilarity relation iff $\overline{\mathcal{R}}$ is the smallest relation such that*

$$\forall s_1, s_2 \in S.\ \left(s_1 \overline{\mathcal{R}} s_2 \Leftarrow \exists (s_1, a, t_1) \in T.\ (\forall t_2 \in S.\ \forall (s_2, a, t_2) \in T.\ \Rightarrow t_1 \overline{\mathcal{R}} t_2)\right) \wedge \\ (s_1 \overline{\mathcal{R}} s_2 \Leftrightarrow s_2 \overline{\mathcal{R}} s_1)$$

Note that the relation $\overline{\mathcal{R}}$ in the above definition is defined as a least fixed point. Hence, the relation $\overline{\mathcal{R}}$ can be encoded as a predicate `nbisim/2` defined as follows:

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

In the above encoding, `no_matching_context(S2, A, T1)` stands for $(\forall t_2 \in S.\ (s_2, a, t_2) \in T \Rightarrow t_1 \overline{\mathcal{R}} t_2)$, and is in turned defined as:

```
no_matching_context(S2, A, T1) :-
      not trans(S2, A, _)
    ; (findall(T2, trans(S2, A, T2), L),
       all_nbisim(T1, L)).  % T1 is not bisimilar to any T2 in L

all_nbisim(_, []).
all_nbisim(T1, [T2|Ts]) :- nbisim(T1, T2), all_nbisim(T1, Ts).
```

Using `nbisim/2`, the largest bisimulation relation can be written as a predicate `bisim/2` defined as follows:

```
            bisim(S1, S2) :- not nbisim(S1,S2).
```

The above encoding can be executed in XSB (after annotating `nbisim/2` as a tabled predicate and converting `not` in the definition of `bisim/2` to `tnot`) to compute the largest bisimilarity relation. Using this executable specification, we can check for bisimilarity between any two states in an LTS (S, T) in $O(|S| \times |T|)$ assuming unit-time table lookups. The quadratic factor in our encoding comes from checking for bisimulation between (potentially) every pair of states. Considering table lookup times (for looking up `nbisim(S1,S2)`) the complexity can be as high as $O(|S|^3 \times |T|)$ if tables are organized as a list, or $O(|S| \times |T| \times \log |S|)$ if binary tree data structures are used. It should be noted that there are faster bisimulation checking algorithms: the Kanellakis-Smolka algorithm [3] runs in $O(|S| \times |T|)$; Paige and Tarjan's algorithm [7] runs in $O(|T| \times \log |S|)$. These algorithms compute bisimulation classes bottom up and are thus global. In contrast, the goal-direction that results in the quadratic factor also makes our encoding "local": we explore only states needed to prove or disprove the bisimilarity of the two given states. The most important advantage, however, is that encoding can be extended to symbolic bisimulation of value-passing systems.

Before we discuss bisimulation for value-passing systems, we present a modified encoding of `no_matching_context`/3 which will make the structure of the code more transparent in the setting of value-passing.

```
  no_matching_context(S2, A, T1) :-
      forall(bv(S2,A,T1), fv(T2), trans(S2, A, T2), nbisim(T1, T2)).
```

The essential difference is that we have defined a new predicate `forall/4` to eliminate the explicit list-traversal in the original `no_matching_context/3`. The predicate `forall(BV,FV,g,h)` succeeds if the following is true: given the list of bound variables `BV`, for all values of the free variables in `FV`, if `g` is true then `h` is true.

2.2 Bisimilarity in value-passing systems

For value-passing systems, two notions of bisimilarity, namely *early* and *late*, have been identified. Unlike the other actions, input actions in value passing systems can bind variables. The difference between early and late bisimulation arises from this property.

Value-passing systems over infinite domains are inherently infinite. To represent them in a finite fashion, we use symbolic labelled transition systems, as proposed by Hennessy and Lin [2]. In a symbolic LTS, each transition is annotated with a condition, indicating when it can occur. For instance, in Figure 1(b), the transition $(p_{11}(x), f, p_{111})$ is guarded by the condition $x = 0$. Transitions such as $(p_{12}(x), h, p_{121})$ which are not annotated by any condition are implicitly assumed to be guarded by the condition `true`. In general, a transition in a symbolic LTS is a 4-tuple (s, a, g, t), where s and t are the source and target states, a is the action, and g is the condition guarding the transition.

Let s_1, s_2 be two states such that $(s_1, c?x, g_1, t_1)$, $(s_2, c?x, g_2, t_2)$ and $(s_2, c?x, g_3, t_3)$ are transitions in a symbolic LTS. Recall that the states themselves are terms, and the variable x may occur free in states t_1, t_2 and t_3. We say that s_1 *early-simulates* s_2 as long as for each substitution to x which satisfies g_1, either x satisfies g_2 and t_1 early-simulates t_2 or x satisfies g_3 and t_1 early-simulates t_3. Note that the choice of t_2 or t_3 can be made on a per-substitution basis.

We say that s_1 *late-simulates* s_2 only if one of the following hold:

- Every substitution of x which satisfies g_1 also satisfies g_2 and t_1 late-simulates t_2 under this substitution.
- Every substitution of x which satisfies g_1 also satisfies g_3 and t_1 late-simulates t_3 under this substitution.

In other words, the simulation must *uniformly* choose between t_2 and t_3 for all substitutions which satisfy g_1. Thus the difference between the two notions of bisimulation is the nesting order in which transitions and substitutions are quantified (existential and universal, respectively). Instead of explicitly manipulating substitutions, we can encode early and late bisimulations such that the substitutions are maintained and propagated by the logic programming engine, as explained below.

Early Bisimulation: Given a transition (s_1, a, g_1, t_1), there is no matching context if there is one substitution such that either (1) there is no matching transition $(s_2, a, g_2, \_)$ on that substitution, or (2) for every matching transition (s_2, a, g_2, t_2) the destination states t_1 and t_2 are not bisimilar. Let $(s_2, b_1, g_{21}, t_{21}), (s_2, b_2, g_{22}, t_{22}), \ldots,$ $(s_2, b_k, g_{2k}, t_{2k})$ be all transitions from s_2 such that each g_{2i} subsumes g_1 and each b_i subsumes a: i.e., $\exists \theta_i$ such that $g_{21}\theta_i = g_i$ and $b_i\theta_i = a$ for $i \in \{1, 2, \ldots, k\}$. Let σ_i be the substitution under which states t_1 and t_{2i} are not bisimilar. Let $\oplus$ denote the composition of substitutions. Then, $\sigma = \oplus_{1 \leq i \leq k}(\sigma_i\theta_i)$ is a substitution which establishes that there is no matching context. We compute this substitution σ implicitly in XSB using the following encoding. Observe that we represent the extended transition relation of a symbolic LTS by a 4-ary relation `strans/4`.

```
nbisim(S1, S2) :-
        strans(S1, A1, G1, T1), G1,
        no_matching_context(S1, A1, T1, S2).
nbisim(S1, S2) :-
        nbisim(S2, S1).

no_matching_context(S1, A1, T1, S2) :-
        forall(bv(S1,A1,T1,S2), fv(A2,G2,T2),
                        strans(S2, A2, G2, T2),
                        nsimulate).

nsimulate(bv(_,A1,T1,_), fv(A2,G2,T2)) :-
        (G2,
                ( subsumes(A2,A1), nbisim(T1, T2))
                ; not(subsumes(A2,A1))
        )
        ; not(G2).
```

If we compare this with our second encoding of `nbisim/2` for non-value-passing systems, the essential difference is that the call to `nbisim/2` within `forall/4` has been replaced by a more complex relation, `nsimulate/2`. The relation `nsimulate` ensures that if the guard `G2` is "compatible" with the guard `G1` and the action `A2` is "compatible" with the action `A1` in the current context, then `T1` and `T2` are in `nbisim/2`.

Observe that the nested call to `nbisim/2` inherits a new set of constraints from `G1` and `G2`. 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 [2], where the context in which the value-passing bisimulation has to be evaluated 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 nonbisimilarity of the two processes.

Late bisimulation: Given a transition (s_1, a, g_1, t_1), there is no matching context if either (1) there is no matching transition $(s_2, a, g_2, \_)$ on any substitution, or (2) for every matching transition (s_2, a, g_2, t_2), there is a substitution under which the destination states t_1 and t_2 are not 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/2` in `no_matching_context/4` as follows:

```
nsimulate(bv(_,A1,T1,_), fv(A2,G2,T2)) :-
        (G2,
                copy_term((A1,T1,A2,T2), (B1,U1,B2,U2)),
                ( subsumes(B2,B1), nbisim(U1, U2)
                ; not(subsumes(B2,B1))
        )
        ; not(G2).
```

This ensures that each transition from s_2 is evaluated in a separate environment, as required by late bisimulation.

Implementation: We have presented a complete encoding for late and early bisimulation checking for value-passing systems. The encoding can be directly executed in XSB as long as the effect of guards on transitions can be represented finitely using equality (term-unification) constraints. For instance, if the guards test for evenness or oddness, their effect can be simply modeled by facts of the form `even(e(_))` and `odd(o(_))`, and term-equality constraints are sufficient. However, if the guards contain arithmetic inequalities, then the underlying evaluation mechanism should be able to handle constraints. In effect, the above encoding is directly executable in a tabled *constraint* logic programming (CLP) system that can handle constraint domains where the guards in the given LTS can be finitely represented. Moreover, the complexity of the evaluation is $O(|S| \times |T|)$ assuming unit-time table lookup and constraint manipulation, which is same as Hennessy and Lin's procedural algorithm [2]. Furthermore, the encoding (and its implementation) clearly separate the logical aspects of bisimulation from its representational aspects.

We have implemented a metainterpreter for constraint solving that runs over XSB. The metainterpeter maintains the constraint store and simplifies the constraints as they are propagated, thus simulating a tabled CLP environment. Our preliminary experience suggests that one can construct practical systems for checking symbolic bisimulation in a tabled CLP framework.

2.3 Strong- and Weak- Bisimulation

The difference between strong and weak bisimulation arises from the treatment of transitions labeled with internal (τ) actions. Strong bisimulation is defined over the usual transition relation T and treats τ actions as any other. The bisimulation relations we have considered thus far are strong.

Weak bisimulation treats τ actions as unobservable, and does not distinguish between a single τ action and a sequence of τ actions. Weak bisimulation can be defined as follows. Let $R_{\tau^*} \subseteq S \times S$ such that $(s_1, s_2) \in R_{\tau^*}$ if $s_1 = s_2$ or $\exists t$ such that $(s_1, \tau, t) \in T$ and $(t, s_2) \in R_{\tau^*}$; R_{τ^*} is called the τ-closure relation. Let $T_W \subseteq S \times A \times S$ such that $(s_1, a, s_2) \in T_W$ if $a \neq \tau$ and $(s_1, a, s_2) \in T$, or $(s_1, t_1) \in R_{\tau^*}$, $(t_1, a, t_2) \in T$ and $(t_2, s_2) \in R_{\tau^*}$; and $(s_1, \tau, s_1) \in T_W$ if $(s_1, \tau, s_2) \in T$ and $(s_2, s_1) \in R_{\tau^*}$. Weak bisimulation is a bisimulation where the transition relation T is replaced by the relation T_W. Clearly, the computation of relations R_{τ^*} and T_W can be readily encoded and evaluated using tabled resolution. Replacing `strans` in the encoding of early and late bisimulation with the predicate corresponding to

T_W yields weak-early and weak-late bisimulations respectively. Computing weak bisimulations adds the cost of computing $R_{\tau*}$ which, in an XSB encoding, will take $O(|S| \times |T|)$ time assuming unit-time table lookups.

3 Conclusions

In this abstract we demonstrated how the power and versatility of tabled logic programming can be used for testing 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. A detailed performance analysis and extensions to subsequent works of Lin and other researchers are some interesting directions for the future.

Another interesting extension of this work would be to pursue the connection between checking bisimulation for value-passing systems and bisimulation for timed systems. When we represent a value-passing system as a finite symbolic LTS, we are effectively quotienting the infinite state-space of the underlying system into a finite set of equivalence classes. The well-known *region construction* for timed systems, introduced by Alur and Dill [1], is also based on similar intuition. It seems likely, therefore, that the techniques developed for checking bisimulation for value-passing systems should also be applicable to timed systems.

References

[1] R. Alur and D. Dill. The theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

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

[3] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, 1990.

[4] 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.

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

[6] R. Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

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

[8] 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.

[9] 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.