Symbolic Partial Order Reduction for Rule Based Transition Systems

Ritwik Bhattacharya[1], Steven German[2], and Ganesh Gopalakrishnan[1,⋆]

[1] School of Computing, University of Utah
{ritwik, ganesh}@cs.utah.edu
[2] IBM T.J. Watson Research Center
german@watson.ibm.com

Abstract. *Partial order (PO) reduction methods* are widely employed to combat state explosion during model-checking. We develop a partial order reduction algorithm for rule-based languages such as Murphi [4] based on the observation that for finite-state systems, independence conditions used for PO reduction can be encoded as boolean propositions and checked using SAT methods. Comparisons against static-analysis based PO reduction algorithms have yielded encouraging results.

1 Introduction

Partial order (PO) reduction helps combat state explosion by avoiding redundant interleavings [3] among *independent* transitions [12,6,10], generating a representative subset of all interleavings. Traditional PO reduction algorithms rely on syntactic methods (e.g. based on occurences of shared variables) to compute the independence relation. Unfortunately, in the presence of complex data structures like records and arrays, such as is common with cache coherence protocols encoded in languages such as Murphi [4] and TLC [9], these algorithms do not work well — even if concurrent accesses to these aggregate structures occur at disjoint sites. By conducting a deeper semantic analysis based on Boolean SAT methods, one can overlook such 'false sharings' and achieve PO reduction. This short paper sketches our explicit enumeration model checking algorithms for PO reduction that benefit from a SAT-based analysis for independence.

There has been extensive research on partial order reduction methods [3]. Few previous works address reduction for formalisms without processes. Partial order reduction algorithms have also been proposed for symbolic state exploration methods [1]. The algorithm there is based on a modified breadth first search, since symbolic state exploration is essentially breadth first. The *in-stack check* of the traditional partial order algorithm is replaced by a check against the set of visited states. An alternative to the traditional runtime ample set computation algorithm is discussed in [8].

⋆ Supported in part by NSF Award ITR-0219805 and SRC Contract 1031.001.

D. Borrione and W. Paul (Eds.): CHARME 2005, LNCS 3725, pp. 332–335, 2005.

2 Partial Order Reduction

Two transitions are *independent* if, whenever they are enabled together at a state, (i) firing either one does not disable the other (*enabledness*), and (ii) firing them in either order leads to the same state (*commutativity*). A transition is *invisible* with respect to a property if it does not change the truth values of any of the atomic propositions occurring in the property. The ample-set method proceeds by performing a modified depth-first search where, at each state, a subset of all the enabled transitions is chosen, called the *ample set*. Transitions from the ample set are then the only ones pursued from that state. This leads to a subset of the entire state space being explored. It is important to ensure that for each path in the full graph, there is a *representative* path in the reduced graph. The following conditions, adapted from [3], guarantee the existence of such representative paths: **C0 :** An ample set is empty if and only if there are no enabled transitions. **C1:** Along every path in the full state graph that starts at a state s, the following must hold - if there is an enabled transition that depends on a transition in the ample set, it is not taken before some transition from the ample set is taken. **C2 :** If a state is not fully expanded, then every transition in the ample set is invisible. **C3**[1]**:** There is at least one transition in every ample set that leads to a state not on the current dfs stack, which ensures that at least one transition in the ample set does not create a cycle.

3 Implementing PO Reductions for Murphi

We compute the independence relation by encoding the *enabledness* and *commutativity* relations as boolean propositions, and using a SAT solver to conservatively check them. First, we take the code fragments defining the guards and actions, and transform them into equivalent Lisp S-expressions. These are then combined to form S-expressions representing the *enabledness* and *commutes* relations for each pair of transitions, which are symbolically evaluated to produce formulas over finite data types. We do this over the entire syntax of Murphi, handling loops (by unrolling), procedures, and functions in the process. To check commutativity, for example, the SAT solver is given a formula of the form $g_1(S) \wedge g_2(S) \Rightarrow t_1(t_2(S)) \neq t_2(t_1(S))$ for an arbitrary S (perhaps unreachable — this being the source of conservativeness). If satisfiable, t_1 and t_2 are potentially non-commuting; otherwise, they are commuting. The invisibility checks can similarly be encoded as boolean formulas and symbolically evaluated.

Constructing the Ample Set: Our algorithm for constructing the ample set is shown in Figure 1. Line `2` picks an enabled, invisible transition (called the *seed transition*) at each state, and tries to form an ample set using this transition. Once a seed transition has been chosen, lines `5-7` compute the transitive closure of the ample set with respect to the dependence relation. Lines `11-15` check for a violation of the **C1** condition. If there is no violation, lines `16-19` check whether at least one of the transitions in the ample set leads to a state not on the current

[1] For a proof of the sufficiency of this form of the condition see [7].

```
1  proc ample(s) {
2   ample := { pick_new_invisible(enabled(s)) };
3   if (empty(ample))
4    return enabled(s);
5   while (exists_dependent(enabled(s),ample)) {
6    ample := ample + all_dependent(enabled(s),ample);
7   }
8   non_ample := all_transitions \ ample;
9   if ((ample = enabled(s)) or exists_visible(ample))
10   return enabled(s);
11  for (t_d in disabled(s))
12   if (dependent(t_d, ample))
13    for (t_o in non_ample)
14     if (t_o != t_d and !leavesdisabled(t_o,t_d))
15      return enabled(s);
16  for (t_a in ample) {
17   if (!(t_a(s) in onstack(s)))
18    return ample;
19  }
20  return enabled(s);
21 }
```

Fig. 1. Ample set construction algorithm for Murphi

stack. If this is the case, we return this ample set. Otherwise, we return the set of all enabled transitions.

4 Results and Conclusions

Our algorithms have been implemented in the **POeM** tool [2], which extends Murphi. We have run **POeM** on examples of varying sizes, and the results are shown in Table 1. Significant reduction is achieved in a number of the examples, the most dramatic being the dining philosophers benchmark labeled DP in the table, where, for 10 philosophers, there is over 99% reduction. The symbolic PO algorithm always does better than the static algorithm in our examples, in terms of the number of states generated. GermanN refers to German's cache protocol for N nodes. It currently yields insignificant reductions because of the existence of transitions dependent only in unreachable states. We are working on strengthening the guards with local invariants, to restrict the independence checks to reachable states.

Instead of SAT, better results might be obtained through higher level decision procedures for quantifier free formulas with equality, finite arithmetic and arrays [11,5], especially given the possibility of initially representing Murphi procedures and functions using uninterpreted functions.

Table 1. Performance of partial order reduction algorithm

Example	Unreduced		Static PO		Symbolic PO	
	States	Time	States	Time	States	Time
Bakery	33	0.14	33	0.14	21	0.14
Burns	82010	2.65	82010	5.02	81542	8.76
Dekker	100	0.17	100	0.17	90	0.17
Dijkstra4	864	0.29	864	0.29	628	0.31
Dijkstra6	11664	0.62	11664	0.88	6369	0.98
Dijkstra8	139968	6.65	139968	13.15	57939	35.32
DP4	112	0.22	112	0.22	26	0.22
DP6	1152	0.27	1152	0.27	83	0.25
DP10	125952	13.85	125952	17.27	812	0.34
DP14	>20000	>60	>20000	>60	7380	1.4
Peterson2	26	0.15	26	0.15	24	0.15
Peterson4	22281	0.3	22281	0.53	14721	0.58
German3	28593	0.43	28593	0.78	28332	1.31
German4	566649	31.15	566649	39.9	562542	72.43

References

1. Rajeev Alur, Robert K. Brayton, Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Partial-order reduction in symbolic state space exploration. In *Computer Aided Verification*, pages 340–351, 1997.
2. R. Bhattacharya. http://www.cs.utah.edu/formal_verification/poem-0.4.tar.gz.
3. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, December 1999.
4. David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design*, pages 522–525, 1992.
5. C. Flanagan, R. Joshi, X. Ou, and J.B. Saxe. Theorem Proving Using Lazy Proof Explication. In *Computer Aided Verification*, pages 355–367, 2003.
6. Patrice Godefroid. Using partial orders to improve automatic verification methods. In *Computer Aided Verification*, pages 176–185, 1990.
7. G.J. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proc. 12th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, Orlando, Fl., June 1992.
8. R. Kurshan, V.Levin, M.Minea, D.Peled, and H. Yenigün. Static partial order reduction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98)*, pages 345–357, 1998.
9. Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Pearson Education, Inc., 2002.
10. Doron Peled. All from one, one for all: On model checking using representatives. In *Computer Aided Verification*, pages 409–423, 1993.
11. Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A Cooperating Validity Checker. In *Computer Aided Verification*, pages 500–504, 2002.
12. Antti Valmari. A stubborn attack on state explosion. In *Computer Aided Verification*, pages 156–165, 1990.