Exploiting Symmetry and Transactions for Partial Order Reduction of Rule Based Specifications*

Ritwik Bhattacharya[1], Steven M. 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. Rule based specifications are popular for specifying protocols, such as cache coherence protocols specified in TLA+, Murphi, or the BlueSpec language. Specifications in these notations are a collection of unordered *rules* of the form *guard* → *atomic_updates*. There is no notion of a sequential process with local scope or specialized communication channels, and each rule tends to update multiple fields of the global state. It is believed that partial order (PO) reduction, a powerful state space reduction technique, is difficult to achieve in such a setting. Partial order reductions attempt to visit a smaller set of states by selectively exploring a subset of all enabled transitions at each state, based on the *independence* of transitions. In earlier work, we have reported a suitable algorithm for this purpose, where the independence relation is computed using symbolic analysis and SAT. In this paper, we expand on this algorithm and show how to exploit some commonly seen characteristics of rule based specifications. First, many of these systems have a *transactional* nature, such as the request/grant transactions of cache coherence protocols. We show how to use this information while picking subsets of transitions at each state. Second, many of these systems are *parameterized*, and also exhibit *symmetry*. We show that, for such systems, the SAT-based computation of the *independence* relation between rules can be performed once and for all in a manner that is accurate for *all* parameterized instances of the protocol. Third, we show that sharpening the SAT-based independence computation through local invariants can aid PO reduction. Here, we propose a way by which users may *guess* these invariants: we can check these invariants *and* the property of interest in one combined phase under PO reduction (we prove that there is no circularity in this process). Our results indicate that with the above measures, rule based systems can have efficient and effective PO reduction algorithms.

* This work was supported in part by NSF Award CCR-0219805, and SRC Contract 2005-TJ-1318.

A. Valmari (Ed.): SPIN 2006, LNCS 3925, pp. 252–270, 2006.

1 Introduction

Rule-style specification of protocols are widely employed. They are often written in languages such as Murphi [6], TLA+ [14], or BlueSpec [1], and are often used for modeling cache coherence, file systems, message networks, and solutions to similar locking/concurrency problems. Protocols specified in these notations are a collection of unordered rules of the form $guard(state) \rightarrow atomic\_updates$. The state of systems modeled in such notations can be a global aggregate datatype such as a record of arrays of simpler types or other records. There is no notion of a sequential process with local scope or FIFO channels[1], and each rule tends to update multiple fields of the global state. Such specifications are natural to write for domains such as cache coherency, where designers prefer a declarative approach to modeling using an unordered collection of rules.

Hardware systems and protocols often have a very large degree of concurrency, and it is natural to view this concurrency in terms of collections of rules, where many rules may update the same part of the global state space. A similar effect can be achieved in process based paradigms such as Promela/SPIN [10], for example by implementing each rule as a process. However, where the division of state variables into global and local components is either not apparent, or not possible, the process based approach has no advantages, and presents the same difficulties and challenges to state space reduction as rule based systems.

Also, such unordered collection of rules are often automatically compiled into the underlying cache coherency engine [1]; from this perspective, the unordered and declarative nature of the rules leads to concurrent hardware that can be modularly understood.

SAT Based Independence Computation, Exploiting Local Invariants: Given these differences in specification style, however, it is clear that one of the main weapons to combat state explosion of these protocol models during enumerative model checking—namely *partial order reduction* (PO reduction or POR)—becomes difficult to realize for rule based systems. Partial order reductions [9, 22] are based on avoiding redundant interleavings that are explored by explicit state enumeration model checkers to preserve concurrency semantics. Computing the *independence relation* over transitions is a crucial aspect of partial order techniques. Independence of a pair of transitions formalizes the notion that they don't interfere with each other's enabledness at a state, and result in the same state no matter which order they're executed in. In general, the greater the number of pairwise independent transitions in a system, the greater is the reduction achievable using partial order techniques.

Traditional partial order reduction algorithms rely on a syntactic check of transitions for references to (global) state variables, in order to compute the

[1] While FIFO channels are convenient for modeling, and help obtain the benefits of PO reduction, (i) rule based languages we know about do not support channels, and (ii) designers often want something other than any one of the standard varieties of channels such as FIFO, sorted, lossy, etc., such as reordering queues in the case of the Wildfire protocol [15].

independence relation. The rationale is that transitions that refer to disjoint sets of state variables can safely be marked independent of each other. However, in the presence of high-level data structures, especially arrays, this approach can be overly conservative. For example, consider the two guard::action pairs below:

1. $((i >= 0) :: a[i+1] :=\text{True})$
2. $((i <= 0) :: a[i+2] :=\text{False})$

A syntax-based PO reduction approach (such as used in SPIN [10]) would classify rules 1 and 2, which both access the same array variable, as dependent. Our symbolic simulation based approach would, on the other hand, determine that the rules are independent, since in all states where both rules are enabled, the rules access different indices of the array a. In particular, our independence computation based on SAT will, as most used definitions of independence require (e.g., [5, Chapter 10]), (i) start the system from a general symbolic state s (all states are broken into the individual bits), (ii) pick a pair of rules r_1 and r_2, (iii) determine whether, whenever r_1 and r_2 are enabled in s, firing one rule leaves the other enabled, and (iv) determine whether $r_1(r_2(s)) = r_2(r_1(s))$ ($r_1(s)$ denotes the state that results from executing the action of rule r_1 at state s). All this is performed by symbolically simulating the actions of rules, and then performing SAT checks on the resulting propositional expressions. For example, the last step is realized by seeing whether $r_1(r_2(s)) \neq r_2(r_1(s))$ can be satisfied.

Obviously, doing this analysis starting from a general starting state can result in pessimal independence information. Later we show how *local invariants* can help sharpen the analysis to states that contain reachable states but exclude certain unreachable states. One of our main results is that the kinds of invariants that tend to give sharp results regarding independence are *not* those that we are proving at the top level (e.g., cache coherence), but those that (i) provide some information about which rules might follow which other rules, and (ii) those that tend to capture relationships between different global variables. More specifically, we observe from the available list of benchmark protocols written by others that one often employs more variables than necessary in a protocol for modeling convenience. Our observation is that often it is necessary to pin down relationships that might exist between these variables. In any case, once a local invariant g_1 is obtained, a rule such as $g \rightarrow action$ is strengthened into $g \wedge g_1 \rightarrow action$. We show that such strengthened rules yield a better (larger) independence relation.

Clearly, we do not wish that the solving the problem of POR lead to another hard problem, namely that of invariant discovery! Here, we have found a simpler, but often equally effective approach. We show a method by which designers can (i) guess these invariants (even if they are incorrect, we will be safe, as we show below), (ii) perform the independence check with respect to modified rules of the form $g \wedge g_1 \rightarrow action$, *but* (iii) while performing model checking using our POR algorithm (that of course enjoys the benefit of the larger independence relation), be checking not merely *original_property* but actually *original_property* $\wedge (g \Rightarrow g_1)$. If there are more guards than one strengthened in this way, for each such g and its strengthening g_i, we would have the conjunct $g \Rightarrow g_i$ in the top-level property being verified. This way, if any of the g_i excludes a state that g includes, it will be detected while verifying the modified property that includes $g \Rightarrow g_i$. A similar approach is presented in [20], but they only consider invariants related to synchronization.

Symmetry and Carry-over of Independence Computation: We observe that many of the protocol descriptions in languages such as Murphi employ scalarsets [12]. Scalarsets are a specialized data type that, by restricting the types of operations permitted on elements of the type, guarantee *symmetry* of systems with respect to permutations over elements of the scalarset type. Furthermore, the variables that are of scalarset type participate in *ruleset* constructs inside protocols. Rulesets of Murphi model parametric sets of rules. For example, if a cache coherence protocol has N nodes, it is quite likely that there is a ruleset collectively modeling each node's behavior. In the Murphi model checker, rulesets defined over scalar set parameters are handled as follows: (i) the user picks a number (*e.g.* 4) for the size parameter of the rule set, (ii) the model-checker creates four copies of the rules, and (iii) while model checking, these four rules are non-deterministically fired in all possible ways (this is necessary, as each rule instance may be, for example, modeling the behavior of a different caching node), (iv) after each state is generated, Murphi's symmetry algorithm performs *state canonicalization* [12], thus generating representative states out of each.

Our approach to exploit the scalarset symmetry is as follows: we demonstrate that under certain conditions, we can, for the purposes of SAT-based independence computation, analyze an instance of size N but model-check an instance $M > N$. This way, the number of rules analyzed as well as the data structures involved in the analysis (e.g., if the data structure sizes were determined by N) would be much smaller. Later, the user may model-check an instance M that is much bigger than N, because currently parameterized proofs are hard to obtain and the designer takes the approach of flushing out bugs in as high an instance as they can. However, in model checking the M-sized instance, the user can employ the same independence relation as calculated on the basis of an N-sized instance.

Ample Set Computation Exploiting Transactions: In *ample set* based POR algorithms (e.g., [5, Chapter 10]), computing the independence relation is only part of the story; ample set formation is a run-time activity where independence comes into play. The search algorithm computes at each state in the search space a set of transitions called an ample set. The ample set is a subset of the enabled transitions, and the search algorithm only generates next states via the transitions in the ample set. Ample sets are minimal (subject to certain sufficient conditions) subsets of the set of enabled transitions at a state, and act as a locally optimal heuristic to maximize global state space reduction. Naturally, smaller ample sets are preferred over larger ones. In our approach, ample sets are computed by picking a *seed* transition and performing a least-fixed-point computation based on the dependence relation. We then check to ensure that the set thus obtained satisfies a set of sufficient conditions **C0 - C3**[5, Chapter 10] (see Appendix A). Picking the seed transition turns out to be an important factor in forming small ample sets, and we have discovered that the *transactional* nature of many of the systems modeled using the rule-based paradigm allows for a particularly effective choice of the *seed transition*. These transactional systems often operate in fairly sequential phases of requests, intermediate processing, and grants. Thus, detecting the phase of the transaction in progress allows us to pick, as seed transitions, transitions that will take the current transaction forward, in effect delaying the "scheduling" of new transactions as long as possible. The sequential nature of transactions means that only a very few (often just one) transitions are enabled at any given point in the transaction, resulting in small ample sets.

Roadmap: Section 2 goes over the notations and definitions. In Section 3, we introduce the notion of exploiting scalarset symmetry, and how it can be used to extrapolate independence results for parameterized systems. We also state and prove our main theorem regarding this result. Section 4 discusses the use of *transactions* to form more effective ample sets, and Section 5 proposes a novel technique called *guard strengthening* to soundly refine rule-based systems to achieve higher independence between transitions. Section 6 discusses experiments and results. Related work and conclusions are in Section 7.

2 Background, Notations and Definitions

A labeled finite state transition system $\mathcal{F}$ is a 5-tuple $\langle S, T, I, P, L\rangle$ where S is a finite set of *states*, T is a finite set of deterministic *transitions*, such that every $t \in T$ is a partial function $t : S \mapsto S$, $I \subseteq S$ is the set of initial states, P is a set of atomic propositions, and $L : S \mapsto 2^P$ labels each state with a set of propositions that are true in the state. Without loss of generality, we assume that T includes a transition from every state to itself. A *labeled path* of a finite state system is an infinite sequence starting with a state and then alternating transitions and states,

$$s_0, t_0, s_1, t_1, s_2, t_2, \ldots,$$

where $\forall i \geq 0 : t_i(s_i) = s_{i+1}$. A labeled path is called a labeled *run* if it starts with a state in I. For any labeled path p of a system, we define the predicate $\mathbf{before}(p, t_1, t_2)$ to be true when t_1 occurs before the earliest occurrence of t_2 in p, or t_2 does not occur in p. Let the set of all labelled paths of a finite state system be $\mathcal{P}$. The *restriction* of $\mathcal{P}$ with respect to a state s, written $\mathcal{P}_{|s}$, is the set of all labelled paths in $\mathcal{P}$ starting from the state s. A transition t is said to be *enabled* at a state s if $\exists s' \in S : t(s) = s'$. We define the predicate $\mathbf{en}(s, t)$ that is true exactly when t is enabled at s. We also define the predicate $\mathbf{enabled}(s) = \{t \in T \mid \mathbf{en}(s, t)\}$. Two transitions t_1 and t_2 are *independent* iff the following conditions hold:

- *Enabledness*(En): $\forall s \in S : \mathbf{en}(s, t_1) \wedge \mathbf{en}(s, t_2) \Rightarrow \mathbf{en}(t_1(s), t_2) \wedge \mathbf{en}(t_2(s), t_1)$
- *Commutativity*(Co): $\forall s \in S : \mathbf{en}(s, t_1) \wedge \mathbf{en}(s, t_2) \Rightarrow t_1(t_2(s)) = t_2(t_1(s))$

We define the predicate $\mathbf{ind}(t_1, t_2)$, that is true exactly when t_1 and t_2 are independent, and $\mathbf{dep}(t_1, t_2) = \neg\mathbf{ind}(t_1, t_2)$. Independence is a symmetric, irreflexive relation. A *property* π of a system is a formula in next-time free *linear* *temporal logic* [17] (LTL_{-X}), such that the set of propositions in the logic is P. In this paper, however, we restrict our attention to *invariant* properties, which are properties that must hold at every reachable state of the system. For any property π, we define $\mathbf{props}(\pi) \in 2^P$ as the set of propositions occurring in π. A transition t is *invisible* with respect to a property π, written as $\mathbf{inv}_\pi(t)$, iff:

$$\forall s_1, s_2 \in S : t(s_1) = s_2 \Rightarrow L(s_1) \cap \mathbf{props}(\pi) = L(s_2) \cap \mathbf{props}(\pi)$$

Our partial order reduction algorithm has two phases, *static* and *dynamic*. In the first, static phase, we compute the truth values of the **dep** relation for each pair of transitions of the system. For rule based systems (and Murphi in particular, on which our implementation is based), as mentioned earlier, transitions are described as *rules*, each of which is a guard/action pair. To compute the **dep** relation for a pair of rules, we symbolically simulate the effect of each rule's action, to build propositional expressions representing the enabledness and commutativity conditions. Murphi also allows the parametric definition of *rulesets*, as mentioned earlier. When computing the **dep** relation for a pair of rules from rulesets, we employ various techniques to avoid the combinatorial explosion that would occur if we were to actually instantiate every parameterized rule with every possible value. For example, for a pair of rules from different rulesets, we build propositional expressions that leave the symbolic variables representing the parameters totally unconstrained. When checked by a SAT solver, this effectively corresponds to checking every pair of instances of the two rules. Of course, this is conservative, because if the SAT solver finds a satisfying assignment to one of these expressions, it only means that two *particular* rule instances (corresponding to the values assigned to the parameter variables by the satisfying assignment) are dependent. Our algorithm, however, will mark *every* pair of rule instances dependent, in this case. Similar techniques are also applied when computing the **dep** relation for rules from the same ruleset, as well as for a single rule and a rule from a ruleset.

When the static phase terminates, we have obtained a complete **dep** relation, for every pair of rule instances of the system. In the dynamic phase of POR, ample sets are constructed at each state visited, during a *depth first* traversal of the state graph. At each state, we pick an arbitrary enabled transition called *seed*, and form an ample set around it, as follows. We first obtain the *dependency closure* of all transitions that are dependent on *seed*. Now we are left with *enabled independent* transitions, and *disabled dependents* (clearly, another possible category, namely *disabled independent* transitions are completely inconsequential for ample-set formation). We have to ensure (as condition C1 of [5, Chapter 10] requires) that there is no transition in the *unreduced* state graph such that one of these disabled dependent transitions could fire **before** one of the transitions in the current ample set. This could easily happen if one of the *enabled independent* transitions could fire and "wake up" one of the disabled dependent transitions. We have experimented with two schemes, the second of which gives better performance:

Approach 1: **If** the *disabled dependent* set is empty, **then** the ample set is the dependency closure set (thus it leaves out the enabled independent transitions), **else** all enabled transitions are in ample.

Approach 2: **If** there are any *disabled dependent* transitions, ensure that they can fire *only* as a result of any one of the transitions in the dependency closure set firing (thus precluding that they may occur **before** one of the ample set transitions). This information can be computed and stored in the static phase of the POR algorithm, thus avoiding a run-time cost.

In Section 4, we discuss how these seed transitions are picked according to the weighing scheme described earlier. We do have implementations of the other checks, namely C0, C2, and C3, as [5] requires. In particular, for the C3 condition which avoids *ignoring*, we have an implementation that implements an on-the-fly *in-stack* check. Details of these checks are omitted, but can be found in [3].

3 Computing Independence for Parametric Systems

A Murphi system description is considered to be *parameterized* if the state variables and transitions are indexed over one or more parameters. In this paper, we only consider parameterized systems with one parameter. In the absence of a general method for verification of parameterized systems, it is common to verify a system for multiple instances of the parameter. However, realistic system sizes increase greatly with an increase in parameter size, and so does the complexity

```
CONST
  num_clients : 3;
TYPE
  message : enum{empty, req_shared, req_exclusive,
                 invalidate, invalidate_ack,
                 grant_shared, grant_exclusive};
  cache_state : enum{invalid, shared, exclusive};
  client: scalarset(num_clients);
VAR
  channel1: array[client] of message;
  cache: array[client] of cache_state;
RULESET cl: client do
  RULE "client requests shared access"
    cache[cl] = invalid & channel1[cl] = empty ==>
    BEGIN channel1[cl] := req_shared END;
  RULE "client requests exclusive access"
    (cache[cl] = invalid | cache[cl] = shared )
     & channel1[cl] = empty ==>
    BEGIN channel1[cl] := req_exclusive END;
END;
```

Fig. 1. A simple parameterized Murphi system outline

of computing the independence relation. We show that for such parameterized systems, it is sufficient to compute the independence relation for a small parameter size. Model checking can be performed for higher parameter sizes using this independence relation.

As a simple example, consider the Murphi system outline of Figure 1. This is an extract from the parameterized German protocol, and shows the two transitions responsible for making new requests for access to a cache line, in either the shared or exclusive mode. The parameter of this system is the number of clients, represented by `num_clients`. The first thing to note is that there are actually multiple instances of each transition (rule) in the system, for any value of `num_clients`. In this case, there are 3 instances of each rule, corresponding to the range of the variable `cl`. Theoretically, therefore, we need to check 9 pairs of rules for dependence (each of the 3 instances of the first rule against each of the 3 instances of the second rule). However, as we show in [3], it suffices to check a pair where the indices have the same value, and a pair where they have different values, and conservatively extrapolate the results to all pairs. So in the given system, we might choose to check rule $1[\texttt{cl} \leftarrow\ cl_1]$ (the instance of the first rule with `cl` set to cl_1) against rule $2[\texttt{cl} \leftarrow\ cl_1]$ (a pair with the same index value), and rule $1[\texttt{cl} \leftarrow\ cl_1]$ against rule $2[\texttt{cl} \leftarrow\ cl_2]$ (a pair with different index values). Here, cl_1 and cl_2 are symbolic values that are left unconstrained in the propositional expressions that are passed to the SAT solver. In the first case, if the SAT solver is unable to find a satisfying assignment, this implies that none of the rule instances with the same parameter value are dependent, since we left the parameter value unconstrained. In the second case, our algorithm conjoins to the propositional expressions representing enabledness and commutativity, a clause that constrains cl_1 and cl_2 to be different from each other. Thus, if the SAT solver is unable to find a satisfying assignment, it is evident that none of the rule instances with different parameter values are dependent on each other.

What does the truth of the above checks for one value of `num_clients` tell us about the truth of the corresponding checks for a different value of `num_clients`? Consider first the case of the two rules with the same value for the ruleset parameter `cl`. Obviously, these two rules are dependent, for they pass neither the enabledness check nor the commutativity check. This is because they are essentially requests from the same node, and therefore each request disables the other. In this case, it would not make any difference what the range of the index variable `cl` was, since the rules do not count the range in any manner, and only refer to state variables that are directly indexed over `cl`, which we have already instantiated to a particular value. Therefore, for these particular rules, it is sufficient to check independence for a particular instance, to be able to conclude that for every instance of the parameterized system, instantiations of these rules with the same index value will never be independent. Now consider the case of the two rules with different values for the parameter `cl`. In this case, the rules involve entirely disjoint sets of state variables, and hence the rules are independent. However, this is true as long as the values of `cl` are different for the two rules, irrespective of what *particular* values they are. Therefore, here too, it is sufficient to check independence for one instance, and infer independence for all instances.

In the following sections, we develop a framework for describing independence among rules of parameterized systems, and show that under certain assumptions, independence of rules is indeed unaffected by parameter size. Section 3.1 introduces the notation and definitions used, and in Section 3.2 we state and prove the main theorem that relates independence of rules across different instances of a parameterized system.

3.1 Notations and Definitions

Recall that a *scalarset* variable in Murphi is a variable such that the system description is completely symmetric with respect to permutations of the elements of the domain of the scalarset variable. A parameterized Murphi specification, with a single scalarset parameter N, can be described in terms of a first order language over the set of variables of the specification. Following Pnueli *et al*'s notion of *bounded data systems* [18], we partition the set of variables into three broad classes, as follows:

- $\mathcal{V}_1 = \{x_1, x_2, \dots x_a\}$ where x_i is interpreted over $\mathbb{B}$, the boolean domain, and $a \in \mathbb{N}$, the set of natural numbers.
- $\mathcal{V}_2 = \{y_1, y_2, \dots y_b\}$ where each y_i is a *scalarset* variable interpreted over the integer subrange $[1 \dots N]$, and $b \in \mathbb{N}$.
- $\mathcal{V}_3 = \{ar_1, ar_2, \dots ar_c\}$ where each ar_i is an array with index type $[1 \dots N]$, each array's cell type is interpreted over $\mathbb{B}$, and $c \in \mathbb{N}$.

The terms of the language are the boolean constants **True** and **False**, variables of type $\mathcal{V}_1$ or $\mathcal{V}_2$, and array references of the form $ar_i[y_j]$, where $y_j \in \mathcal{V}_2$ and $ar_i \in \mathcal{V}_3$. The valid atomic formulas of our language are partitioned into the set of *ordinary* atomic formulas $\mathcal{O}$ and *quantified* atomic formulas $\mathcal{Q}$, where: $\mathcal{O} = \{x_i \mid x_i \in \mathcal{V}_1\} \cup \{ar_i[y_j] \mid ar_i \in \mathcal{V}_3, y_j \in \mathcal{V}_2\} \cup \{y_i = y_j \mid y_i, y_j \in \mathcal{V}_2\}$, and $\mathcal{Q} = \{\forall x \in 1 \dots N. ar_i[x] \mid ar_i \in \mathcal{V}_3\} \cup \{\forall x \in 1 \dots N. \neg ar_i[x] \mid ar_i \in \mathcal{V}_3\} \cup \{\exists x \in 1 \dots N. ar_i[x] \mid ar_i \in \mathcal{V}_3\} \cup \{\exists x \in 1 \dots N. \neg ar_i[x] \mid ar_i \in \mathcal{V}_3\}$. The set of formulas is then the standard extension of the atomic formulas using the boolean connectives $\wedge$, $\vee$ and $\neg$. We say that the set of all formulas over a set of variables $\mathcal{V}$, $\mathcal{L}(\mathcal{V})$ is the *language* of our logic.

A Murphi system description, which consists of a set of variable declarations, and a set of transitions (rules) defined as *guard/action* pairs, can be mapped into our first order language by mapping the variable definitions to the variables of the language, mapping guards to formulas of the language, and mapping actions as sets of substitutions of variables by terms or formulas of the language. Since the variables in $\mathcal{V}_1$ and $\mathcal{V}_3$ are of boolean type, they can be assigned any valid formula of the language, because Murphi allows arbitrary boolean expressions as rvalues in assignments. For a complete description of the allowed substitutions, and the corresponding Murphi constructs, see [2]. A state of a Murphi system can thus be seen as an interpretation of the variables of the logic. We denote the set of all states of a system as S, and, in particular, the set of all states of a *parameterized* system with parameter N as $S(N)$.

In bounded data systems, both states (interpretations) and the satisfaction of formulas, are *symmetric* with respect to any permutation of the indices [18]. This is enforced in Murphi syntax by declaring the parameter range to be a *scalarset* type.

3.2 The Carry over Theorem

We now show that in the above setting, we can compute the dependence relation between transitions for all parameter sizes $N > 1$, by computing the relation for a small size, calculated as described below.

Enabledness: Given a pair of rules $\langle g_1, a_1 \rangle$ and $\langle g_2, a_2 \rangle$, they satisfy the enabledness condition when:

$$g_1(s) \wedge g_2(s) \Rightarrow g_1(a_2(s)) \wedge g_2(a_1(s)) \tag{3.1}$$

is valid over $S(N)$, the set of all states (interpretations) s, $g_1(s)$ denotes the evaluation of the formula g_1, given the interpretation s of the variables, $a_1(s)$ (with a slight abuse of notation) denotes the application of the substitutions represented by a_1 to the variables, followed by an evaluation of the resulting terms over the interpretation s. Similarly for g_2 and a_2.

We would like to find a bound, $\widehat{N}$, such that 3.1 is valid over $S(N)$ for all N, $N > 1$ iff it is valid over $S(N)$ for all N, $1 < N \leq \widehat{N}$. To arrive at such a bound, we proceed as follows: in formula 3.1, we push negations inside atomic formulas of type $\mathcal{Q}$ (ie, a formula $\neg\forall i.ar_j[i]$ is converted into the equivalent formula $\exists i.\neg ar_j[i]$, and so on for every atomic formula of type $\mathcal{Q}$). Let the cardinality of the set $\mathcal{V}_2$ be k, the number of *existentially quantified* atomic formulas of type $\mathcal{Q}$ in a guard g_i be e_i, and the number of *universally quantified* atomic formulas of type $\mathcal{Q}$ in g_i be u_i.

Theorem 1. *3.1 is valid over $S(N)$ for all N, $N > 1$ iff it is valid over $S(N)$ for all N, $1 < N \leq \widehat{N}$, where $\widehat{N} = k + 2 \times (\max\{e_1, e_2\} + \max\{u_1, u_2\})$.*

Proof (sketch)[2]: To show this, it is sufficient to show that the negation of 3.1:

$$g_1(s) \wedge g_2(s) \wedge (\neg g_1(a_2(s)) \vee \neg g_2(a_1(s))) \tag{3.2}$$

is satisfiable for $N > \widehat{N}$ iff it is satisfiable for some N, $1 < N \leq \widehat{N}$. To show this, moreover, it is sufficient to show that if 3.2 is satisfiable over $S(N)$, for $N > \widehat{N}$, it is satisfiable over $S(\widehat{N})$.

By counting the number of existentially quantified atomic formulas of the forms $\exists x.ar_i[x]$ or $\exists x.\neg ar_i[x]$ in 3.2, which can be equivalently written as $ar_i[p]$ and $\neg ar_i[q]$ respectively, where p and q are fresh variables of type $\mathcal{V}_2$, we can show that the total number of variables of type $\mathcal{V}_2$ is bounded by $\widehat{N}$. Thus, given an interpretation s over $S(N)$ that satisfies 3.2, it can assign at most $\alpha \leq \widehat{N}$ different values to these variables. Without loss of generality, assume

[2] The proof is very similar to, and follows closely, the proof of **Claim 3** in [18, Section 4].

that these values are $v_1 < v_2 < \cdots < v_\alpha$. Since the system is symmetric, there is a permutation over the indices $[1..N]$ that maps v_k to k, for every $k \in 1..\alpha$. Let $\tilde{s}$ be the state derived from s by applying this permutation-induced transformation to the set of variables above. Clearly, $\tilde{s}$ is also an interpretation that satisfies 3.2. To construct the interpretation $\hat{s} \in \widehat{N}$ that satisfies 3.2, we let $\tilde{s}$ and $\hat{s}$ agree on the interpretation of the variables in V_1 and V_2. For the remaining variables $ar_1, ar_2, \ldots ar_c$, we let $\tilde{s}$ and $\hat{s}$ agree on the values of all $ar_i[k]$, for $k \le \alpha$. After replacing existentials by new variables, the formula 3.2 is a formula over the variables in V_1 and V_2, and universally (over the parameter size) quantified expressions over V_3. Since $\tilde{s}$ and $\hat{s}$ agree on the interpretation of all of the above, $\hat{s}$ satisfies 3.2 over $S(\widehat{N})$. By similar reasoning, we can also compute bounds on the commutativity condition. Taken together, these provide an overall bound on the size of the system for which independence checks need to be performed for partial order reduction.

4 Transaction-Based Priorities for Ample Set Construction

For many of the kinds of systems that are typically described using the rule-based paradigm (e.g., protocols of various types), it is often the case that the system proceeds along fairly sequential paths called *transactions*. For example, consider a typical directory-based cache coherence protocol. Most activities in such protocols begin with the cache controller making a request for a line. This request travels to the directory controller which typically evicts other caches from the sharing group by sending invalidations. Thereafter, the directory controller sends the line back to the requesting node. Modeled in Murphi, we can say that (i) this whole activity consists of a *transaction* (refer to [16] for somewhat related notions of a transaction), and (ii) there are Murphi rules that begin a transaction, there are rules that are somewhere in the "middle" of transactions (e.g., invalidation rules), and finally there are rules that end transactions. One can often obtain the situation of a rule—whether it is at the beginning, middle, or end of a transaction—through concrete execution on small instances of the protocol. Most designers also clearly know the situation of rules within a transaction. Clearly, a transaction can involve actions of multiple components (the requesting node and the directory controller in the example above). This is a slightly different notion than the notion of a transaction as a sequence of actions within a single thread (or component), as presented in [19], for example. In any case, we *weigh* each rule as follows: (i) rules that begin transactions are weighed "low," (ii) the rules that end transactions are weighed "high," (iii) rules that are in the middle of transactions are weighed "medium." We need not be exact in how we assign numeric values to "low, medium, and high." Users can be completely wrong in these weight assignments—the only consequence being poorer ample sets but never incorrect execution.

Given a set of weights for the transitions, we use them to pick the *seed* transition during ample set computations. Enabled transitions that have the highest priority are picked as seed transitions at each state. Effectively, this results in "scheduling" ongoing transactions with greater priority, and postponing the start of new transactions as long as possible. Note that this is completely sound, because we will only be able to postpone the start of a new transaction as long as the transition that starts it is independent of transitions that belong to the ongoing transaction. The results of applying this heuristic while computing ample sets are discussed in Section 6.

5 Strengthening Guards

It is often the case that a pair of rules is independent at all reachable states, but dependent at some unreachable state(s). Our analysis, as described so far, marks such rules dependent, since it starts from an entirely general symbolic state. To be able to use the independence of these rules during partial order reduction, a simple idea is to find potential *strengthenings* for guards, that don't change the enabledness of rules in reachable states, and extend the independence of rules to *all* states, both reachable and unreachable. This is useful while checking the **C1** condition, since the fewer the dependent transitions, the smaller the likelihood of there being disabled transitions dependent on the ample set.

To actually discover these strengthenings requires a deep understanding of the protocol involved, and we discuss some intuitions in Section 6.

Once we have strengthened the guards of transitions, it is necessary to show that these strengthenings are sound, and do indeed preserve the semantics of the original transitions. We now show that it is sufficient to model check the strengthened system with a modified property, to be able to prove the soundness of the strengthenings.

Since Murphi transitions are guard action pairs (g_i, a_i), strengthening the guards corresponds to adding predicates p_i to the guards of transitions t_i. Define the *strengthening operator* Θ over transitions such that:

$$\Theta(\langle g_i, a_i \rangle) = \begin{cases} \langle g_i \wedge p_i, a_i \rangle & \text{if } t_i \text{ is strengthened} \\ \langle g_i, a_i \rangle & \text{otherwise} \end{cases}$$

We extend Θ to apply to runs $\sigma = \langle s_1, t_1, s_2, \dots, s_k, \dots \rangle$ so that $\Theta(\sigma)$ results in the sequence (not necessarily a run) $\langle s_1, \Theta(t_1), s_2, \dots, s_k, \dots \rangle$. Let the original system be $\mathcal{F}$, and the modified system $\mathcal{F}'$. Note that both systems have the same set of states, and the same initial state predicate I. Assume that the property to be verified of the original system was P. We model check the new system with the property $P \wedge Str$, where:

$$Str = (g_{i_1} \rightarrow p_{i_1}) \wedge (g_{i_2} \rightarrow p_{i_2}) \wedge \dots \wedge (g_{i_k} \rightarrow p_{i_k})$$

$g_{i_1} \dots g_{i_k}$ are the k guards of the original system that have been strengthened with the predicates $p_{i_1} \dots p_{i_k}$.

Definition. A run $\sigma = \langle s_1, s_2, \dots, \rangle$ of a system satisfies an invariant property P, written as $\sigma \models P$, iff the property is true at every state in the run.

Definition. A system $\mathcal{M}$ satisfies an invariant property P, written as $\mathcal{M} \models P$, iff every run of the system satisfies the property.

Theorem 2.
$$\mathcal{F}' \models P \wedge Str \Rightarrow \mathcal{F} \models P$$

Proof: By contradiction. Assume that the antecedent of the theorem is true, and assume that there is a run $r = \langle s_1, t_1, s_2, t_2, \dots, s_m, \dots \rangle$ of $\mathcal{F}$ that does not satisfy the property P. Without loss of generality, we assume that $s_1, s_2, \dots s_{m-1} \models P$, and $s_m \not\models P$. If $\Theta(r)$ is a run of $\mathcal{F}'$, $s_m \models P \wedge Str$, which implies that $s_m \models P$, contradicting our assumption. Therefore, assume that $\Theta(r)$ is not a run of $\mathcal{F}'$. Then, there is a t_k, such that $\Theta(t_k)$ is not enabled at s_k, and $\Theta(\langle s_1, t_1, s_2, \dots s_k \rangle)$ is a valid prefix of a run of $\mathcal{F}'$, such that $s_1, s_2, \dots s_k \models P \wedge Str$. Since r is a run of $\mathcal{F}$, t_k is enabled at s_k.

Case 1: $\Theta(t_k) = \langle g_k, a_k \rangle$. In this case, since $\Theta(t_k)$ is not enabled at s_k, this implies that $s_k \not\models g_k$. But we know that t_k is enabled at s_k. That is, $s_k \models g_k$, leading to a contradiction.

Case 2: $\Theta(t_k) = \langle g_k \wedge p_k, a_k \rangle$. In this case, we have $s_k \not\models g_k \wedge p_k$. However, we know that $s_k \models g_k$. Also, $s_k \models P \wedge Str$. That is, $s_k \models g_k \rightarrow p_k$. Therefore, $s_k \models g_k \wedge p_k$, leading to a contradiction.

Thus, in every case, we arrive at a contradiction, and hence, the theorem is true, and, by model checking the strengthened system for the property $P \wedge Str$, we can prove that the strengthenings of the guards are sound.

If the model check fails, on the other hand, we have to manually examine the error trail to determine whether the property failed, or whether one of the strengthenings does not hold, and rerun the model check after making the necessary changes, in the latter case.

6 Experimental Results, and Analysis

We have run **POeM** on a number of examples of different sizes, and Table 1 shows our overall results on some mutual exclusion algorithms and a cache coherence protocol. *The experiments in Table 1 were performed with Murphi's symmetry reduction turned on, whenever scalar sets were employed.* This is safe because symmetry and partial order reductions are orthogonal to each other, and can be combined for safety property verification[7]. Guard strengthening and transaction-based weights were not employed in these examples, and are discussed later. In the table, the columns under "Unreduced" represent the number of states explored, and the time taken for the verification to complete, without any partial order reduction. The columns under "Static PO" represent the same figures for the case where a static, syntax-based analysis was used to determine the independence relation (this was our initial prototype version of POeM before we moved on to the use of SAT for independence computation). The columns under "Symbolic PO" represent the figures for **POeM**. The final column, "Analysis Time", is the time taken by **POeM**'s symbolic evaluation based module to compute the independence relation. In cases where we've used the

Table 1. Performance of partial order reduction algorithm

Example	**Unreduced**		**Static PO**		**Symbolic PO**		**Analysis**
	States	**Time**	**States**	**Time**	**States**	**Time**	**Time**
Bakery	157	0.1	157	0.1	119	0.1	8.9
Burns	82010	1.83	82010	3.65	69815	11.67	52.9
Dekker	100	0.13	100	0.13	90	0.13	11.6
Dijkstra6	11664	0.57	11664	0.88	4900	1.17	17.9
Dijkstra8	139968	4.32	139968	8.81	33286	8.98	*CO*
Dijkstra10	>1.5M	>1000.0	>1.5M	>1000.0	202248	82.6	*CO*
DP6	1152	0.32	1152	0.36	90	0.31	0
DP10	125952	7.44	125952	7.86	823	0.48	0
DP14	>1.0M	>800.0	>1.0M	>800.0	7395	2.6	0
Peterson2	26	0.15	26	0.15	24	0.15	4.6
Peterson4	22281	0.3	22281	0.53	14721	0.58	*CO*
German6	7378	1.31	7378	1.36	2542	0.83	32.4
German8	42717	14.6	42717	15.23	10827	4.6	*CO*
German10	193790	127.24	193790	131.83	36606	24.91	*CO*
Leader1	683	0.32	683	0.36	21	0.10	8.5
Leader2	12651	0.20	12651	0.33	12651	0.33	4.6

carry over theorem of Section 3, the entry for higher instances is marked *CO*, and represents the fact that the results were carried over from the analysis of the smallest instance in the table. The experiments were run on a dual processor Xeon 3GHz machine with 1GB of RAM. As can be seen, **POeM** is most effective on large examples, where the overhead of performing the symbolic analysis, and computing an ample set at each state, is outweighed by the savings that result from a far fewer number of states being explored.

Assessing Guard Strengthenings: We experimented with guard strengthenings on the German protocol, as well as the Stanford FLASH [13] cache coherence protocol, and these results are now discussed. The German cache coherence protocol is a directory-based protocol for maintaining coherence among shared memory multiprocessors, proposed by Steven German [8] The Murphi description of the protocol only models a single address/cache line, and a parameterized number of processors.

Our technique for generating predicates to strengthen guards is to first run **POeM** directly on the protocol, and analyze the resulting dependency matrix. For pairs of rules that **POeM** marks dependent, we examine the test(s) that failed (enabledness, dependency, or both), and try to reason about predicates that, if added, would make the rules independent, without violating the properties we wish the protocol to hold. If we are able to come up with such predicates, we add them to the guard, and add the corresponding implication predicate to the invariant to be proved.

Run directly on the German protocol, **POeM** concludes that the rule “home sends invalidate message” is dependent on the rule “home sends reply to client-exclusive”.

The guards for the two rules are:

```
rule "home sends invalidate message"
    (home_current_command = req_shared & home_exclusive_granted
     | home_current_command = req_exclusive)
    & home_invalidate_list[cl]
    & channel2_4[cl] = empty

rule "home sends reply to client -- exclusive"
    home_current_command = req_exclusive
    & client_requests[home_current_client]
    & forall i: client do home_sharer_list[i] = false endforall
    & channel2_4[home_current_client] = empty
```

It is evident that the two rules ought never to be enabled together, and therefore, marked independent. However, it is not apparent what predicate is to be added to enforce this. It is clear from the existing guards that, if the rules are to be simultaneously enabled, `home_current_command = req_exclusive` must be true. Looking at the rule "home picks new request", which sets this variable, leads to the realization that, in the case of a request for exclusive access, the home node copies the `home_sharer_list` to the `home_invalidate_list`. The protocol then clears an entry in the invalidate list once it sends out the invalidate message to that client, and clears the entry in the sharer list once the client has sent the acknowledgment to the invalidate. This means that, at the time the home node sends out an invalidate message to a client, that client must be on the sharer *and* invalidate lists. Therefore, we can add the predicate `home_sharer_list[cl]` to the guard for the rule "home sends invalidate message":

```
rule "home sends invalidate message"
    (home_current_command = req_shared & home_exclusive_granted
     | home_current_command = req_exclusive)
    & home_invalidate_list[cl]
    & channel2_4[cl] = empty
    & home_sharer_list[cl]
```

Similar reasoning is used to strengthen the guards of other pairs of rules that we determine to have been falsely marked dependent by **POeM**.

As is evident, the ability to effectively strengthen the guards of a given protocol depends on a good understanding of the workings of the protocol, which we possess for the German protocol. From a practical perspective, however, industrial design groups possess a deep understanding of their protocols, and we have reasons to believe that designers will, when presented with $false$ entries in the independence matrix, be able to identify guard strengthenings as discussed above. Our results on the FLASH coherence protocol further demonstrate the effectiveness of this approach, yielding over 60% reduction for 4 nodes, although, with over 30 rules, and many auxiliary variables, it is a much more complex protocol and guard strengthenings were only performed for the most obvious cases.

Table 2. Advantages of Guard Strengthening

Example	Without Strengthening				With Strengthening			
	Unreduced		POeM		Unreduced		POeM	
	States	Time	States	Time	States	Time	States	Time
German6	13270	2.68	4485	1.29	7378	1.42	2542	0.74
German8	81413	30.28	20104	8.87	42717	14.5	10827	4.56
German10	378236	260.96	69613	49.04	193790	126.6	36606	24.72
FLASH	6336	0.78	6336	1.46	2888	0.46	2146	0.64

Table 2 shows the results of running the protocols with and without strengthened guards.

Assessing Transaction-Based Priorities: The next set of experiments run on the German protocol were aimed at testing the significance of user-defined priorities for rules, over the automatically computed priorities, which are based on the number of variable references. Table 3 shows the comparison between the two methods of assigning priorities to rules. Lower weights translate into a higher priority for the rule to be picked as the seed transition.

The user defined priorities were assigned in such a fashion as to give higher priority to rules that complete transactions, the transactions in this case being the requests for exclusive or shared access to a line. Rules that represent the intermediate steps of a transaction were given medium priority, and rules that represent the start of a transaction were given the lowest priority.

In the case of the German protocol, user-defined priorities gave a distinct performance boost to **POeM**, resulting in upto an 80% reduction over the already reduced state space explored by **POeM** using the regular variable reference count based priorities. This confirms our intuitions that user-defined priorities are a good way to select the seed transition around which to form ample sets.

Recently [4], we have built an experimental variant of Murphi that records the sequence of rules fired with respect to user-identified *request* rules and *completion* rules. From this experimental version of Murphi, we observe that rule weights can be computed with reasonable accuracy based on concrete executions of small instances of the protocol.

Table 3. Transaction-based priorities vs. Variable reference based priorities (G=German)

Ex	Without Strengthening				With Strengthening			
	Trans. based wts		Var based wts		Trans. based wts		Var based wts	
	States	Time	States	Time	States	Time	States	Time
G6	2521	0.66	4485	1.29	1166	0.36	2542	0.74
G8	8098	2.75	20104	8.87	2851	0.94	10827	4.56
G10	20968	10.52	69613	49.04	5890	2.57	36606	24.72

6.1 Protocols That Yield Low Reductions with POeM

Our partial order reduction algorithm yields large reductions on many complex protocols, but also fails to yield significant reductions on some others. An example in Table 1 is the leader election protocol from [5](Leader2). This example is of a network of nodes in a ring topology running an algorithm to determine the node with the largest id. The algorithm involves exchanging messages through buffers, and POeM's independence computation, relying on the *primary* and *alternate* checks, concludes that all the message-passing transitions of each node are dependent on those of *all* of the other nodes, although each node's transitions only depend on its *neighbor*'s transitions (since neighboring nodes read and write a common message buffer). This indicates that rule-based systems might benefit from making buffers/queues first-class data structures in their language, allowing partial order reduction algorithms to take advantage of the orchestrated fashion in which these buffers operate. This example also lacks scalarset symmetry, and it might be possible to improve our algorithm by examining other kinds of specialized symmetries, such as the ring symmetry of this example. On the other hand, POeM is very successful on the other leader election example studied (Leader1), since that example employs a single-cell buffer, and thus forces the nodes to proceed in lock-step fashion. This also makes the case that the specification style can often influence the amount of reduction achievable.

7 Conclusions and Future Directions

In this paper, we have described in detail a number of heuristics and techniques to further improve the efficiency of partial order reduction algorithms for rule-based systems, and demonstrated the advantages of our heuristics over conventional, static analysis based partial order reduction algorithms for these types of systems.

An interesting experiment to perform might be to replace our SAT-based backend with a more powerful solver such as CVC [21], which combines decision procedures for fragments of arithmetic, theories for uninterpreted functions, etc. It might also be possible to automatically generate candidate predicates to strengthen guards, based on the satisfying assignment returned by the SAT solver/decision procedure, in case two rules are found to be dependent.

References

1. Arvind. Bluespec: A language for hardware design, simulation, synthesis and verification, Invited Talk. In *MEMOCODE*, pages 249–249, 2003.
2. Ritwik Bhattacharya. http://www.cs.utah.edu/~ritwik/carryover.html.
3. Ritwik Bhattacharya, Steven German, and Ganesh Gopalakrishnan. A symbolic partial order reduction algorithm for rule based transition systems. Technical Report UUCS-03-028, School of Computing, University of Utah, 2003.
4. Xiaofang Chen. personal communication.

5. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* MIT Press, December 1999.
6. David Dill. The Stanford Murphi Verifier. In *CAV '96*, pages 390–393, 1996.
7. E. Allen Emerson, Somesh Jha, and Doron Peled. Combining partial order and symmetry reductions. In *TACAS '97*, pages 19–34, 1997.
8. Steven German. http://www.cs.utah.edu/~ritwik/poem/german_cache.m.
9. Patrice Godefroid. Using partial orders to improve automatic verification methods. In *CAV '90*, pages 176–185, 1990.
10. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.
11. 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*, 1992.
12. C. Norris Ip and David L. Dill. Better verification through symmetry. In *CHDL '93*, pages 87–100, 1993.
13. J. Kuskin and D. Ofelt et al. The Stanford FLASH multiprocessor. In *SIGARCH94*, pages 302–313, May 1994.
14. L. Lamport. Specifying concurrent systems with TLA+. In *Calculational System Design*, 1999.
15. Leslie Lamport. The wildfire challenge problem. http://research.microsoft.com/users/lamport/tla/wildfire-challenge.html.
16. Vladimir Levin, Robert Palmer, Shaz Qadeer, and Sriram K. Rajamani. Sound transaction-based reduction without cycle detection. In *SPIN '05*, pages 106–122, 2005.
17. Amir Pnueli. A temporal logic of concurrent programs. In *Theoretical Computer Science*, pages 45–60, 1977.
18. Amir Pnueli, Sitvanit Ruah, and Lenore D. Zuck. Automatic deductive verification with invisible invariants. In *TACAS '01*, pages 82–97, 2001.
19. S. Qadeer, S. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, 2004.
20. Scott D. Stoller and Ernie Cohen. Optimistic synchronization-based state-space reduction. In *TACAS '03*, pages 489–504, 2003.
21. Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A Cooperating Validity Checker. In *CAV '02*, pages 500–504, 2002.
22. Antti Valmari. A stubborn attack on state explosion. In *CAV '90*, pages 156–165, 1990.

APPENDIX

A Sufficient Conditions for Ample Set Construction

Adapted from [5, Chapter 10], the sufficient conditions **C0-C3** for constructing valid ample sets are:

– **C0 :** $\forall s \in S : \mathbf{ample}(s) = \phi \Leftrightarrow \mathbf{enabled}(s) = \phi$. An ample set is empty if and only if there are no enabled transitions.

- **C1:** $\forall s \in S : \forall t_1, t_2 \in T : t_1 \in \mathbf{ample}(s) \wedge t_2 \notin \mathbf{ample}(s) \wedge \mathbf{dep}(t_1, t_2) \Rightarrow \forall p \in \mathcal{P}_{|s} : \exists t_3 \in \mathbf{ample}(s) : \mathbf{before}(p, t_3, t_2)$ Along every path in the full state graph that starts at 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 :** $\forall s \in S : \mathbf{ample}(s) \neq \mathbf{enabled}(s) \Rightarrow \forall t \in \mathbf{ample}(s) : \mathbf{inv}_\pi(t)$. If a state is not fully expanded, then every transition in the ample set is invisible.
- **C3**[3] **:** $\forall s \in S : \mathbf{ample}(s) \neq \mathbf{enabled}(s) \Rightarrow \exists t \in \mathbf{ample}(s) : t(s) \notin \mathbf{onstack}(s)$ There is no transition t that is enabled in a state that is part of a cycle, and is not in the ample set of any state in that cycle.

[3] For a proof of the sufficiency of this form of the condition see [11].