

State Reduction Using Reversible Rules ^{*}

C. Norris Ip David L. Dill

Department of Computer Science
Stanford University, Stanford, CA 94305, U.S.A.
Email: {ip,dill}@cs.stanford.edu

Abstract

We reduce the state explosion problem in automatic verification of finite-state systems by automatically collapsing subgraphs of the state graph into abstract states. The key idea of the method is to identify state generation rules that can be inverted. It can be used for verification of deadlock-freedom, error and invariant checking and stuttering-invariant CTL model checking.

1 Introduction

Formal verification methods that rely on state enumeration are very effective in catching errors in designs. However, such methods suffer from the state explosion problem: the vast number of possibilities cannot be explored within available time and memory. The number of possibilities usually grows exponentially with number of components in the system.

Although many techniques (e.g. BDDs [1, 3]) have been developed to tackle the state explosion problem, a lot of practical designs are still too complicated for automatic verification, especially for high level systems or protocols [8]. In this exposition, we identify certain regularities in the state space of a typical message-passing protocol and use them to generate a reduced state space for verification. The reduced state space can be explored using any conventional verification algorithms, such as explicit state enumeration or symbolic model checking. In particular, we use the explicit state enumeration algorithm in the Mur φ verifier [4] to generate the results presented.

The new method converts a state space to a reduced state space by firstly identifying a subset of state generation rules as reversible rules, and then collapsing every subgraph connected by the reversible rules into an abstract state. The abstract state is represented by a unique state of the subgraph, called the *progenitor* of the subgraph.

The properties of the reversible rules make sure that every state in a subgraph can be generated from the progenitor via the reversible rules, and each state can be mapped back to the progenitor by reverse execution of the reversible rules. Compared to conventional abstraction [7], our method does not require the user to provide a suitable abstract domain and does not produce false negative results as one often found with badly chosen abstract domains. It can be used for the verification of deadlock-freedom, error and invariant checking, and stuttering-invariant CTL model checking (c.f. [11]).

^{*} Supported by Semiconductor Research Corporation under contract 95-DJ-389. Sun Micro-systems provided computers.

33rd Design Automation Conference, 1996

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation of the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistributed to list, requires prior specific permission and/or a fee.

Our main goal is to reduce the amount of memory used in storing the examined states. In several message-passing protocols, the reductions of state space sizes are from 30% to more than 90%. Less time is used for most of the protocols, also. Our new method also complements the symmetry reduction strategy [9, 2, 5], allowing for additional reductions when the two methods are combined.

2 An Example

In this section, we illustrate our method through a cache coherence protocol [12]. Cache coherence is a way of implementing a shared-memory abstraction on top of a message-passing network. Whenever a processor wants to load a cache entry into its cache, it sends a request to the memory, which keeps track of which processors have read-only copies or writable copies of the cache entry, as well as the associated data.

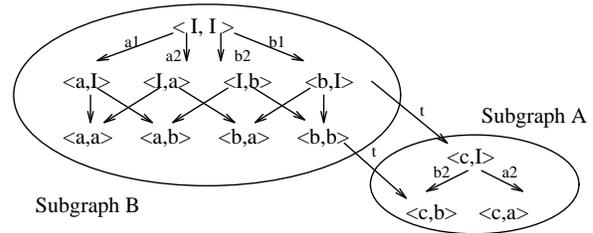


Figure 1: A Motivating Example

Lets take a look at a typical state graph of such a protocol, as shown in Figure 1. In subgraph A, the second processor has an invalid cache entry and no outstanding requests (processor state I), and it has two options: to issue a request for a read-only copy (processor state a) or a writable copy (processor state b). Such transitions are legal regardless of what the other processors and the memory are doing (processor state c).

These transition rules also lead to state explosion: in subgraph B, 9 states are generated from two processors in state I . In general, 3^k states are generated from the state with k processors in the processor state I . If we collapse these subgraphs into abstract states, we can obtain a smaller state graph for verification.

In order to do so, we take advantage of the reversible property of these transition rules: the information in the next state and the transition rule is sufficient to reconstruct the original state. This reversible property allows us to reverse execute the rules to find the progenitor.

3 Background

In this section, we summarize the background for automatic formal verification using state space exploration.

A state graph (a Kripke structure) represents the behavior of a system by capturing all possible sequences of system states. In its simplest form, it is a quadruple $\langle Q, Q_0, \Delta, \mathbf{error} \rangle$, where Q is a set of states, $Q_0 \subset Q$ is a set of initial states, $\mathbf{error} \in Q$ is a unique error state, and $\Delta \subseteq Q \times Q$ is a transition relation such that **error** transits only to **error**.

A state graph can be defined implicitly by a set of *transition rules*, T , where each rule maps a state to a successor state. Formally, for all $q_1, q_2 \in Q$, we have $(q_1, q_2) \in \Delta$ if and only if there exists $t \in T$ such that $q_2 = t(q_1)$.

We usually denote $(q_1, q_2) \in \Delta$ as $q_1 \longrightarrow q_2$, denote $q_2 = t(q_1)$ as $q_1 \xrightarrow{t} q_2$, and denote $q_n = t_n(\dots t_1(q_0)\dots)$ as $q_0 \xrightarrow{t_1 \dots t_n} q_n$. A finite sequence of states q_0, \dots, q_n is called a *path* if and only if $q_0 \longrightarrow q_1 \longrightarrow \dots \longrightarrow q_n$. A state q is *reachable* from p if and only if there exists a path p, \dots, q .

Because of limited space, we state theorems for error and invariant checking only: the system contains an error if and only if **error** is reachable from the initial states. If the reversible rules preserve all atomic propositions in a CTL formula, the reduced state space can be used for stuttering-invariant CTL model checking. To check for freedom from deadlock, every state in the subgraph generated from a progenitor has to be checked explicitly to see if it is a deadlock state.

4 The Abstract State Space

First of all, we define the reversible rule set as follows:

Definition 1 (reversible rule set) *Given a state graph $\langle Q, Q_0, \Delta, \mathbf{error} \rangle$ generated by the transition rules T , the subset $U \subseteq T$ is a reversible rule set if and only if*

- For all $q \in Q, r \in U$, if $q \neq \mathbf{error}$, then $r(q) \neq \mathbf{error}$.
- For all $r \in U$, there exists a function r^* such that for all $q \in Q$, if there exists a unique $q' \in Q$ such that $q' \neq q \wedge r(q') = q$, then $r^*(q) = q'$. Otherwise, $r^*(q) = q$.
- For all $q \in Q$ and $r_1, r_2 \in U$, we have $r_1^*(r_2^*(q)) = r_2^*(r_1^*(q))$. There exists an integer n such that for all $q \in Q$ and $r \in U$, we have $(r^*)^{n+1}(q) = (r^*)^n(q)$.

For every state, the properties of the reversible rules allow us to “undo” the effect of the reversible rules until we reach the progenitor, in which nothing can be undone:

Definition 2 (progenitors) *Given a state graph $\langle Q, Q_0, \Delta, \mathbf{error} \rangle$ generated by transition rules T , and a reversible rule set $U \subseteq T$, a state $q \in Q$ is a progenitor if and only if for every $r \in U$, $r^*(q) = q$. A progenitor q is a progenitor of a state q' if and only if there exist $r_1, \dots, r_n \in U$, such that $q \xrightarrow{r_1 \dots r_n} q'$.*

For example, in a cache coherence protocol, if every reversible rule generates a new request message, a state with no request in the network is a progenitor. The properties of the reversible rules allow us to find the progenitor easily:

Theorem 1 (uniqueness of progenitor) *Given a state graph $\langle Q, Q_0, \Delta, \mathbf{error} \rangle$ generated by transition rules T , a reversible rule set $U = \{r_1, \dots, r_n\} \subseteq T$, and an integer n such that for all $q \in Q$ and $r \in U$, $(r^*)^{n+1}(q) = (r^*)^n(q)$, we have that for all $q \in Q$, the unique progenitor for q is $\theta(q) = (r_m^*)^n(\dots((r_1^*)^n(q))\dots)$.*

We use the progenitor $\theta(q)$ to represent the set of states reachable from $\theta(q)$ via the reversible rule set:

Definition 3 (reduced state graph by progenitors) *Given a state graph $A = \langle Q, Q_0, \Delta, \mathbf{error} \rangle$ generated by transition rules T , and a reversible rule set $U \subseteq T$, the reduced state graph $A_U = \langle \theta(Q), \theta(Q_0), \theta(\Delta), \mathbf{error} \rangle$ is defined as*

- $\theta(Q) = \{\theta(q) | q \in Q\}$ and $\theta(Q_0) = \{\theta(q) | q \in Q_0\}$
- $(q_1, q_2) \in \theta(\Delta)$ if and only if there exist $q \in Q$ and $t \in T \setminus U$ such that $\theta(q) = q_1$ and $\theta(t(q)) = q_2$.

The on-the-fly reduction algorithm for error and invariant checking is shown in Figure 2. Every state in the subgraph represented by a progenitor is generated using a local search in the procedure *Local_Search()*. The next states of each state are generated using the rules in $T \setminus U$, and they are converted to the corresponding progenitors for comparison and storage. The changes from a conventional search algorithm are highlighted by a left vertical bar or an underline.

```

Algorithm_1() {
  Reached = Unexpanded = {  $\theta(q)$  |  $q \in Q_0$  }
  While Unexpanded  $\neq \emptyset$  Do
    Remove a state  $s$  from Unexpanded
    Local_Search(s) }

| Local_Search(state s) {
|   Local_Reached = Local_Unexpanded = {s}
|   While Local_Unexpanded  $\neq \emptyset$  Do
|     Remove a state  $s$  from Local_Unexpanded
|     Generate_Original_Next_States(s);
|     For each transition rule  $r \in U$  Do
|       Let  $s' = r(s)$  in
|         If  $s'$  is not in Local_Reached then
|           Put  $s'$  in Local_Reached and Local_Unexpanded }

Generate_Original_Next_States(state s) {
  For each transition rule  $t \in T \setminus U$  Do
    Let  $s' = t(s)$  in
      If  $\theta(s') = \mathbf{error}$  then stop and report error
      If  $\theta(s')$  is not in Reached then
        Put  $\theta(s')$  in Reached and Unexpanded }

```

Figure 2: Algorithm 1

Algorithm 1 is similar to a state space caching algorithm [10], in which some of the original states generated are not stored in the hash table. However, we discard the states only when it can be determined from the existing states in the hash table that those discarded states have already been examined. Therefore, our method never expands a state more than once.

In the subsequent lemmas and theorems, we use (subscripted) q to represent a state in Q , (subscripted) r to represent a rule in U , (subscripted) t to represent a rule in $T \setminus U$, and (subscripted) k to represent a non-negative integer. Furthermore, we denote a transition in the reduced state graph as $q_1 \xrightarrow{t} q_2$ if there exists $q \in Q$ such that $\theta(q) = q_1$ and $\theta(t(q)) = q_2$.

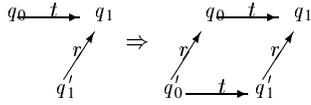
Lemma 1 *Given a state graph $A = \langle Q, Q_0, \Delta, \mathbf{error} \rangle$ generated by transition rules T , and a reversible rule set $U \subseteq T$, if $q_0 \xrightarrow{r_1 \dots r_k} q$ is in A , then $\theta(q_0) \xrightarrow{t} \theta(q)$ is in A_U .*

Theorem 2 (soundness) *Given a state graph $A = \langle Q, Q_0, \Delta, \mathbf{error} \rangle$ generated by transition rules T , and a reversible rule set $U \subseteq T$, if **error** is reachable from the initial states in A , **error** is also reachable from the initial states in A_U .*

In the remainder of this section we show that under an extra condition, called the *essential properties*, the reduction is also complete. The essential properties imply that for every state pair q, q' and every essential rule r such that $r(q) = q'$, if q' is reachable from the initial states, q is also reachable from the initial states.

Definition 4 (essential rule set) Given a state graph $\langle Q, Q_0, \Delta, \mathbf{error} \rangle$ generated by transition rules T , the rule set $U \subseteq T$ is an essential rule set if and only if

- for all $q \in Q_0$, we have $\theta(q) \in Q_0$.
- for all $r \in U, t \in T \setminus U$, if there exist distinct $q_0, q_1, q'_1 \neq \mathbf{error}$ such that $q_0 \xrightarrow{t} q_1$ and $q'_1 \xrightarrow{r} q_1$, then there exists $q'_0 \in Q$ such that $q'_0 \xrightarrow{r} q_0$ and $q'_0 \xrightarrow{t} q'_1$.



Lemma 2 Given a state graph $A = \langle Q, Q_0, \Delta, \mathbf{error} \rangle$ generated by transition rules T , and a reversible and essential rule set $U \subseteq T$, if $q_0 \xrightarrow{t} q_1$ is in A_U , there exist $r_1, \dots, r_k \in U$ such that $q_0 \xrightarrow{r_1, \dots, r_k, t} q_1$ is in A .

Theorem 3 (completeness) Given a state graph $A = \langle Q, Q_0, \Delta, \mathbf{error} \rangle$ generated by transition rules T , and a reversible and essential rule set $U \subseteq T$, if \mathbf{error} is reachable from the initial states in A_U , \mathbf{error} is also reachable from the initial states in A .

The same reduced state graph and the same algorithm shown in Figure 2 can be used for verification using a reversible and essential rule set.

5 Speeding up the Generation of the Abstract State Space

In the previous section, we have discussed how we can reduce the number of states stored. However, every originally reachable state is visited during the local search phase of the algorithm, and the time required to generate the reduced state graph would be longer than the time required to generate the original state graph. In this section, we present a condition which allows great reductions in the amount of local search required to generate the same reduced state graph.

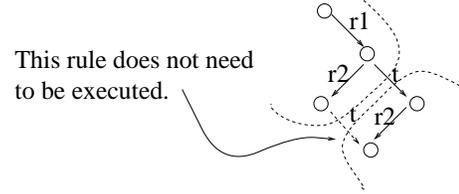
Consider the state graph in Figure 1. There are redundant transitions between subgraph A and subgraph B : the transition t was fired twice, once from $\langle b, I \rangle$ and once from $\langle b, b \rangle$. In this case, it is not necessary to consider $\langle b, b \rangle$ to generate the reduced state graph: every transition from $\langle b, b \rangle$ has a corresponding transition from $\langle b, I \rangle$ or $\langle I, b \rangle$.

The intuition is that the result of a transition often depends only on the immediate execution of *at most one* reversible rule. For example, consider a message-passing protocol in which each transition checks and removes at most one message from a network, and that every reversible rule generates at least one message. If a transition t depends only on the message generated by a reversible rule r_1 , another reversible rule r_2 can be executed or reverse executed without affecting the execution of t . We call this the singularity property of r_1 and r_2 . With this property, it is sufficient to apply the transition rules only to the progenitors and their immediate successors.

Formally, the singularity property is defined as:

Definition 5 (singularity) Given a state graph $\langle Q, Q_0, \Delta, \mathbf{error} \rangle$ generated by transition rules T , and a reversible rule set $U \subseteq T$, the rule set U is singular if and only if for all distinct q_1, q_2, q_3, q_4 such that $q_1 \xrightarrow{r_1} q_2 \xrightarrow{r_2} q_3 \xrightarrow{t} q_4$, we have either $q_1 \xrightarrow{r_1, t, r_2} q_4$ or $q_1 \xrightarrow{r_2, t, r_1} q_4$.

Therefore, for a state q that is neither a progenitor nor an immediate successor of a progenitor, no transition from q needs to be searched:



Lemma 3 Given a state graph $A = \langle Q, Q_0, \Delta, \mathbf{error} \rangle$ generated by transition rules T , and a singular reversible rule set $U \subseteq T$, if $q \xrightarrow{r_1, \dots, r_k, t} q'$, then there exists r_j such that $q \xrightarrow{r_j, t} q''$ and $\theta(q'') = \theta(q')$.

Theorem 4 (fast reduced state graph generation) Given a state graph $\langle Q, Q_0, \Delta, \mathbf{error} \rangle$ generated by transition rules T , and a singular and reversible rule set $U \subseteq T$, we have $\langle q_1, q_2 \rangle \in \theta(\Delta)$ if and only if there exists $t \in T \setminus U$ such that $\theta(t(q_1)) = q_2$ or there exists $r \in U$ such that $\theta(t(r(q_1))) = q_2$.

Hence, the algorithm can be speeded up to the one shown in Figure 3 if we are not checking for deadlock-freedom. In this algorithm, the number of states examined in the local search is greatly reduced. As shown in Table 1, the practical results for an industrial cache coherence protocol (ICCP) confirmed that many fewer states were examined and the verification finished in a much shorter time.

```

Algorithm_2() {
  Reached = Unexpanded = {  $\theta(q) \mid q \in Q_0$  }
  While Unexpanded  $\neq \emptyset$  Do
    Remove a state  $s$  from Unexpanded
    Local_Search(s) }

Local_Search(state s) {
  Local_Reached = {s}
  Generate_Original_Next_States(s)
  For each transition rule  $r \in U$  Do
    Let  $s' = r(s)$  in
    If  $s'$  is not in Local_Reached then
      Generate_Original_Next_States(s) }

Generate_Original_Next_States(state s) {
  For each transition rule  $t \in T \setminus U$  Do
    Let  $s' = t(s)$  in
    If  $\theta(s') = \mathbf{error}$  then stop and report error
    If  $\theta(s')$  is not in Reached then
      Put  $\theta(s')$  in Reached and Unexpanded }

```

Figure 3: Algorithm 2

While the singularity property and partial order reduction [13, 6] seem to have superficial similarity, there are applications that one does better than the other and neither method dominates the other.

| ICCP: 4 processors | states stored | states examined | time |
|-----------------------------------|--------------------|-----------------|---------|
| Original (with unordered network) | 247,565 | 247,565 | 205s |
| Alg. 1 | 34,005 | 247,565 | 338s |
| Alg. 2 | 34,005 | 123,197 | 128s |
| ICCP: 5 processors | states stored | states examined | time |
| Original (with unordered network) | > 6,500,000 states | | |
| Alg. 1 | 492,075 | 6,568,279 | 4 hours |
| Alg. 2 | 492,075 | 2,206,135 | 66 mins |

Table 1: Comparison of Different Reduction Algorithms

6 Reversible Rules and Symmetry

Reduction using reversible rules is orthogonal to the reduction using symmetry. Symmetry can be defined as an automorphism on the state graph and the transition rules [9]. States p, q are symmetric if there is an automorphism h such that $h(p) = q$. In order to combine symmetry and reversible rules to obtain an even smaller state graph, the reversible rule set must also be symmetric:

Definition 6 (symmetric rule set) *Given a state graph $\langle Q, Q_0, \Delta, \mathbf{error} \rangle$ generated by transition rules T , and a set of automorphisms H , a rule set $U \subseteq T$ is symmetric if and only if for all $r \in U$ and $h \in H$, we have $h(r) \in U$.*

Theorem 5 *A set of automorphisms H on the state graph A is also a set of automorphisms on A_U if U is both symmetric and reversible.*

7 Practical Results

For the results here, the reversible rules were identified manually. However, although not implemented in our verifier yet, these reversible rules can be automatically detected by a combination of programming language design and static analysis of the description of the system, similar to the one in [9].

The verification results for the following protocols are presented in Table 2:

- an industrial directory-based cache coherence protocol (ICCP);
- the Stanford DASH multiprocessor cache coherence protocol (DASHC) and lock protocol (DASHL) [12];
- distributed linked-list protocols (LIST1, LIST2).

For these protocols, a processor typically issues a request on the network, and becomes blocked until a message arrives from the network. Since the messages in the network are received one by one, these transition rules form a symmetric, singular, reversible and essential rule set.

For ICCP, there are five reversible rules for each processor; the space and time reduction obtained are therefore very large. For LIST1 and LIST2, there are two reversible rules for each processor; the space reduction obtained are still quite large. DASHC and DASHL have fewer reversible rules, and the reductions are not as large as the other applications.

The results shown are for systems with a small number of processors only, because we need to be able to generate the original state graphs for comparison purposes. In fact, using the reduction with reversible rules allows us to verify these systems for a much larger size, and also other systems of much higher complexities.

| | ICCP (p4) | LIST1 (p4) | LIST2 (p4) | DASHC (p3) | DASHL (p3) |
|---------------|-----------|------------|------------|------------|------------|
| Original size | 247,565 | 301,029 | 329,601 | 26,925 | 55,366 |
| Size (r) | 34,005 | 112,784 | 162,736 | 15,751 | 36,728 |
| Size (s) | 11,814 | 13,044 | 13,959 | 4,575 | 9,313 |
| Size (s/r) | 1,760 | 4,926 | 6,894 | 2,672 | 6,170 |
| Original time | 205s | 87s | 250s | 114s | 188s |
| Time (r) | 128s | 72s | 239s | 85s | 213s |
| Time (s) | 28s | 13s | 24s | 63s | 96s |
| Time (s/r) | 13s | 11s | 22s | 50s | 96s |

| ICCP | p4 | p5 | p6 | p7 | p8 |
|---------------|---------|--------------------|---------|--------------------|---------|
| Original size | 247,565 | > 6,500,000 states | | | |
| Size (s) | 11,814 | 68,879 | 358,078 | > 1,500,000 states | |
| Size (s/r) | 1,760 | 6,021 | 18,118 | 49,045 | 121,302 |
| Original time | 205s | – | – | – | – |
| Time (s) | 28s | 349s | 3,762s | – | – |
| Time (s/r) | 13s | 98s | 615s | 3,283s | 12,801s |

s : Symmetry Reduction
r : Reversible Rules Reduction
 $p(n)$: n-processor system

Table 2: Practical Results

References

- [1] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. *5th IEEE Symposium on Logic in Computer Science*, 1990.
- [2] E. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *5th International Conference on Computer-Aided Verification*, June 1993.
- [3] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. *Automatic Verification Methods for Finite State Systems*, 1989.
- [4] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.
- [5] E. A. Emerson and A. P. Sistla. Symmetry and model checking. *5th International Conference on Computer-Aided Verification*, June 1993.
- [6] P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, 110(2):305–326, May 1994.
- [7] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. *5th International Conference on Computer-Aided Verification*, April 1993.
- [8] A. J. Hu. *Techniques for Efficient Formal Verification Using Binary Decision Diagrams*. PhD thesis, Stanford University, December 1995.
- [9] C. N. Ip and D. L. Dill. Better verification through symmetry. *11th International Symposium on Computer Hardware Description Languages and Their Applications*, pages 87–100, April 1993.
- [10] C. Jard and T. Jeron. Bounded-memory algorithms for verification on-the-fly. *3rd Workshop on Computer-Aided Verification*, July 1991.
- [11] L. Lamport. What good is temporal logic. *Information Processing 83*, pages 657–668, 1983.
- [12] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH multiprocessor. *Computer*, 25(3), 1992.
- [13] A. Valmari. Stubborn sets for reduced state space generation. *Advances in Petri Nets 1990*, pages 491–515, 1991.