To appear in Int'l Conference on Computer Hardware Description Language, 1993

# Better Verification Through Symmetry[1]

C. Norris Ip and David L. Dill

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

## Abstract

We address the *state explosion problem* in automatic verification of finite-state systems by exploiting *symmetries* in the system description.

We make symmetries easy to detect by introducing a new data type *scalarset*, a finite and unordered set, to our description language. The set of operations on scalarsets are restricted so that states are guaranteed to have the same future behavior, up to permutation of the elements of the scalarsets. We have extended our *Mur$\varphi$ verifier* to generate a reduced state space on-the-fly. The verifier has been applied to cache coherence protocols, reducing the states space searched in verification by over 90%.

In some cases, this method can collapse infinite state spaces into finite spaces. We call this property *data saturation.* Data saturation can be used to exploit data-independence of protocols automatically, without hand-modification of the protocol descriptions.

Keyword codes: D.2.4; F.3.1; F.3.3
Keywords: Program Verification;
Specifying and Verifying and Reasoning about Programs;
Studies of Program Constructs.

## 1 Introduction

Protocols are becoming increasingly important in hardware designs. For example, network and communications protocols are often implemented directly in hardware. As another example, the internal protocols in large multiprocessors are becoming quite complex. Debugging these protocols is a major problem in hardware design.

Automatic methods for verifying finite-state concurrent systems are surprisingly effective at catching design errors. In general, these methods work by enumerating all

[1]This research was supported by the "Multi-Module Systems" thrust of the Stanford Center for Integrated Systems. Sun Microsystems provided the computers.

reachable states of a system [ZWR+80, BWB82, Hol87]. Of course, the major problem with these methods is that the size of the state space may grow very rapidly with the description size. This phenomenon is known as the *state explosion problem.*

We explore a method for reducing the state explosion by exploiting symmetries in the structure of the system to be verified. Structural symmetries induce an equivalence relation between states; for verification, it is sufficient to explore only one state per equivalence class. As a simple example, consider a mutual-exclusion algorithm for two processes, $A$ and $B$. The state where $A$ is in the critical section and $B$ is waiting is, for all practical purposes, equivalent to the state in which $B$ is in the critical section and $A$ is waiting.

The basic idea of exploiting symmetries to reduce the state space in automatic verification is not new. It was described by Huber et al. [HJJJ84] for high-level Petri nets. The idea was further developed by Starke [Sta91] for deadlock and liveness checking in P/T nets. In unpublished work, Clarke, McMillan and Jha at Carnegie Mellon, and (independently) Emerson and Kaufman at the University of Texas have applied the idea to CTL model-checking.

Our goal is to make it trivial to detect and exploit symmetries by inspecting the system description. This is achieved by adding to the description language a new data type, which we call *scalarset.* A scalarset can only be accessed through restricted operations that guarantee certain symmetries to hold on the state graph. The scalarset type has been added to the Mur$\varphi$ protocol description language and verification system developed at Stanford University [DDHY92].

Using the extended Mur$\varphi$ system, we have obtained a reduction of over 90% in the size of state spaces when verifying examples of directory-based cache coherence protocols. The times required to verify the protocols were reduced by more than 40%. The extended system enables us to generate the reduced state space on-the-fly, without generating the original state space. Therefore, less memory is required for the verification.

We have verified cache coherence systems of different sizes. The size of the state space grows exponentially with increasing numbers of processing nodes, but the use of symmetries reduces the degree of blow-up significantly. Furthermore, we have discovered that the new data type makes it easy to exploit a generalized idea of *data independence* in protocols [Wol86], allowing automatic verification of some systems with infinite state spaces.

# 2 Symmetry and the Mur$\varphi$ Verification System

## 2.1 Structural Symmetry

To illustrate the concept of symmetries, let us look at an example of a multiprocessor with caches.

As shown in Figure 1, a multiprocessor system has a number of processing nodes and memory nodes, communicating via an interconnection network. (The system depicted here is a shared-memory multiprocessor which has a directory associated with each memory location. The directory keeps track of which processor has the memory location cached.) Every processor has a distinct name, usually a small integer, called the *processor-id.* However, most of the properties of integers are irrelevant in a high-level

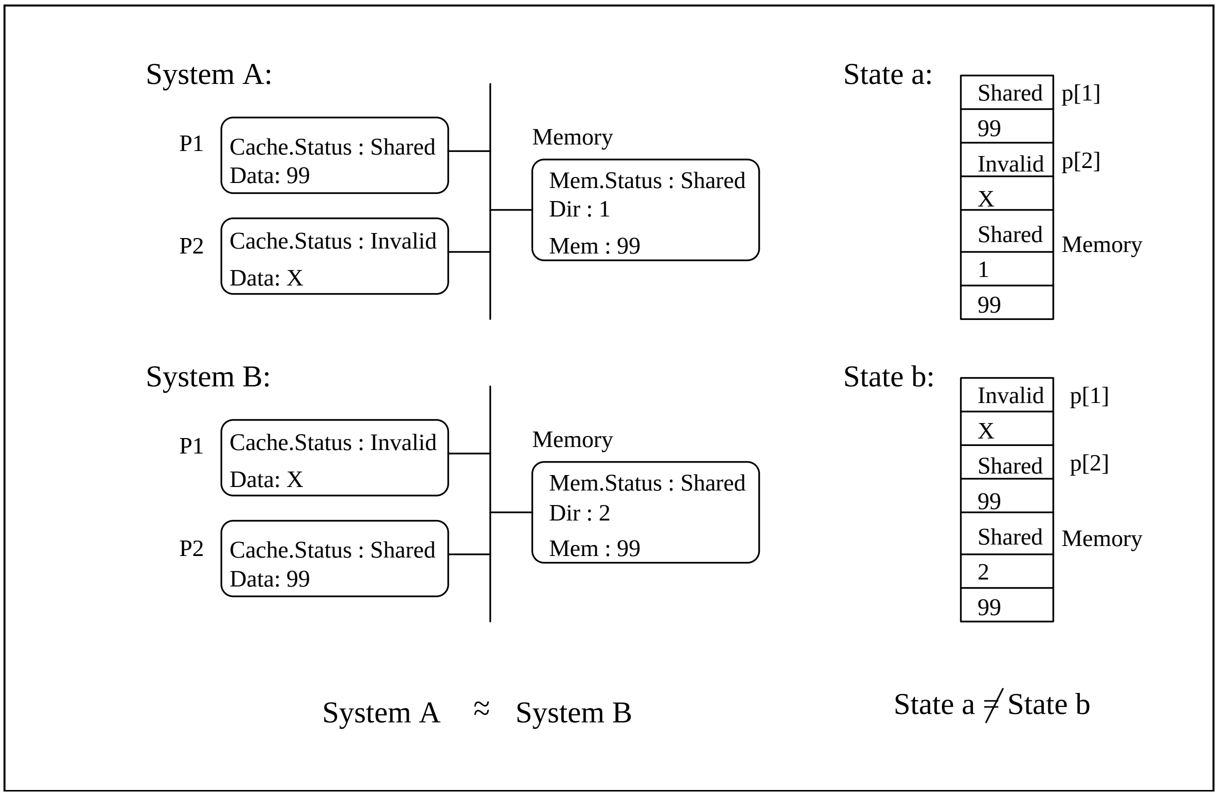


Figure 1: Unnecessary Ordering of Processor Identities

description of the protocol; it only matters whether two processor-ids are the same. It does not matter whether one is numerically less than the other, or whether they are consecutive. But most verifiers have no way to detect this fact, so they may inspect what is basically the same state many times. In addition to processor-ids, there are several other symmetries that hold for the cache coherence protocol example: addresses, data values, memory module-ids and message-ids. Although the numerical properties of addresses and data values are likely to be important at some level of abstraction, they are irrelevant for reasoning about the correctness of the protocol.

If a verifier knows the symmetries of this system, it still may have some difficulty detecting whether two states are equivalent. The states shown in Figure 1 illustrate how the state variables might be laid out in memory in the verifier. Note that, to obtain state $B$ from state $A$ in Figure 1, not only do the processor status entries have to be swapped, but also the processor-ids stored in the memory directory have to be changed accordingly. The problem becomes even more difficult if we consider multiple symmetries at the same time.

We address the two problems of this example:

1. detection of structural symmetries in the system.

2. on-the-fly detection of symmetrically-equivalent states during verification, so that the full state space does not need to be constructed.

```
Type                                      -- a) user defined type
   pid:        1..numProcessor;            -- processor-ids
   mid:        1..numMemory;               -- memory modules-ids
   address:    1..numAddress;              -- address space in a single module
   value:      1..valueCount;              -- possible values in memory word
Var                                       -- b) state variables
   P:          Array [pid] of              -- 1) an array of record storing the status
                 Record                    --    of each processor.
                   State: enum {Invalid, Shared, Master};
                   Value: value;
                 End;
   M:          Array [mid] of              -- 2) a 2-D matrix of records storing the
                 Array [address] of        --    status of each address.
                   Record
                     State: enum {Uncached, Shared_Remote, Dirty_Remote};
                     Dir: Array [1..dirsize] of pid;    -- list of processors using the data
                     Mem: value;
                   End;
   Net:        ...                         -- 3) matrices of records storing messages in
                                           --    the network (details not shown).
```

Figure 2: State Variable Declarations for Cache Coherence Protocol in Mur$\varphi$

## 2.2 The Mur$\varphi$ Verification System

We have extended the Mur$\varphi$ Verification System to achieve our goal of detecting and using symmetry.

The basic Mur$\varphi$ Verification System [DDHY92] consists of the Mur$\varphi$ compiler and the Mur$\varphi$ description language. The Mur$\varphi$ description language is a high-level programming language for the description of finite-state asynchronous concurrent systems. This language was inspired by the Chandy and Misra's Unity language [CM88]. The Mur$\varphi$ compiler generates a C++ program for a particular Mur$\varphi$ description. The C++ program checks the safety properties of the system by explicitly enumerating all reachable states.

A Mur$\varphi$ description consists of four parts:

- Constant, type and variable declarations.
- Procedure declarations and transition rule definitions.
- Start state descriptions.
- Invariant descriptions.

At any time, the system state is specified by the values of the global variables. The rules are guarded commands. As a system executes, a rule is chosen nondeterministically and executed to yield a new system state (by assigning new values to the variables). Although a rule may consist of arbitrarily complex operations, it is executed *atomically*, without interference from other rules in the description. Hence, the use of Mur$\varphi$ leads to

```
Ruleset v : value Do                    – a set of rules for each value
Ruleset h : mid Do                      – a set of rules for each memory module
Ruleset n : pid Do                      – a set of rules for each processor
Ruleset a : address Do                  – a set of rules for each address in a memory module
      Rule "Modifying value at cache"
            P[n].Cache[h][a].State = Locally_Exmod       – if it is an exclusive copy
      ==>
      Begin
            P[n].Cache[h][a].Value := v;                 – then the processor can modify the value
      End;
Endruleset;
Endruleset;
Endruleset;
Endruleset;
```

Figure 3: Transition Rule for Cache Coherence Protocol in Mur$\varphi$

```
Invariant "Only a single master copy exists"
      Forall n1 : pid Do              – for every pair of processors
      Forall n2 : pid Do
      Forall h : mid Do               – for each memory module
      Forall a : address Do           – for each address
         ! ( n1 ≠ n2                          – n1, n2 are distinct, i.e. a real pair of processors
            & P[n1].Cache[h][a].State = Locally_Exmod       – and both have an exclusive copy
            & P[n2].Cache[h][a].State = Locally_Exmod )
      Endforall;
      Endforall;
      Endforall;
      Endforall;
```

Figure 4: Invariant for Cache Coherence Protocol in Mur$\varphi$

an asynchronous, interleaving model of concurrency in which processes interact via shared variables. Examples of Mur$\varphi$ description are shown in Figure 2, 3 and 4.

The types of variables can be arrays, records, integer subranges, Booleans or enumerations. However, the conventional method of coding the system state fails to capture symmetries. For example, let us examine the fragment of a cache coherence protocol description shown in Figure 2. The states are represented by the processor status (variable $P$), the memory status (variable $M$) and the interconnection network (variable $Net$). Because we have used integer subranges (*pid, mid, address* and *value*) for the ids, we have imposed unnecessary ordering among the ids.

## 2.3 Definition of Scalarset

We have introduced a new data type, *scalarset*, to facilitate detection of symmetries and testing of equivalent states. Scalarset supports assignment, testing for equality/inequality,

```
Type                                          -- a) user defined type
    pid:      Scalarset(numProcessor);          -- processor-ids
    mid:      Scalarset(numMemory);             -- memory modules-ids
    address:  Scalarset(numAddress);            -- address space in a single module
    value:    Scalarset(valueCount);            -- possible values in memory word
Var                                           -- b) state variables
    P:        Array [pid] of                    -- 1) an array of record storing the status
                Record                          --    of each processor.
                  State: enum {Invalid, Shared, Master};
                  Value: value;
                End;
    M:        Array [mid] of                    -- 2) a 2-D matrix of records storing the
                Array [address] of              --    status of each address.
                  Record
                    State: enum {Uncached, Shared_Remote, Dirty_Remote};
                    Dir: Array [1..dirsize] of pid;   -- list of processors using the data
                    Mem: value;
                  End;
    Net:      ...                               -- 3) matrices of records storing messages in
                                                --    the network (details not shown).
```

Figure 5: Documenting Symmetry with Scalarsets in Extended Mur$\varphi$

and array indexing. There are no arithmetic operators, no comparison operators other than equality/inequality testing, and no literal constants.

In general, when the numerical values of a subrange are not important, we can convert the subrange to a scalarset, thus enforcing and documenting symmetries that result from permuting the members of the scalarset (Figure 5). In other words, structural symmetry exists whenever a subrange is used only in scalarset operations.

There are four value-binding operations for scalarset variables that preserve symmetries:

1) **Ruleset** *ID: ScalarsetType* **Do** *ruleseq* **Endruleset:** The *Ruleset* construct gives a set of rules that are identical except for the value bound to the variable *ID*. Equivalently, a ruleset nondeterministically chooses a value from the scalarset and binds it to a variable name. Clearly, this operation does not imply any asymmetry among the elements of the scalarset.

2) **For** *ID: ScalarsetType* **Do** *stmtseq* **Endfor:** A for-loop is an iteration over the scalarset values. In order to preserve symmetry, the body of the for-loop is restricted so that the order of execution of the loop-body does not affect the final result. Whenever a variable is modified in one iteration, it cannot be read or modified in another iteration.

3) **Forall** *ID: ScalarsetType* **Do** *booleanexpr* **Endforall:** This operation is the conventional $\forall$ operators on a boolean expression. Because an expression in Mur$\varphi$ has no side effects, properties specified by these constructs are symmetrical.

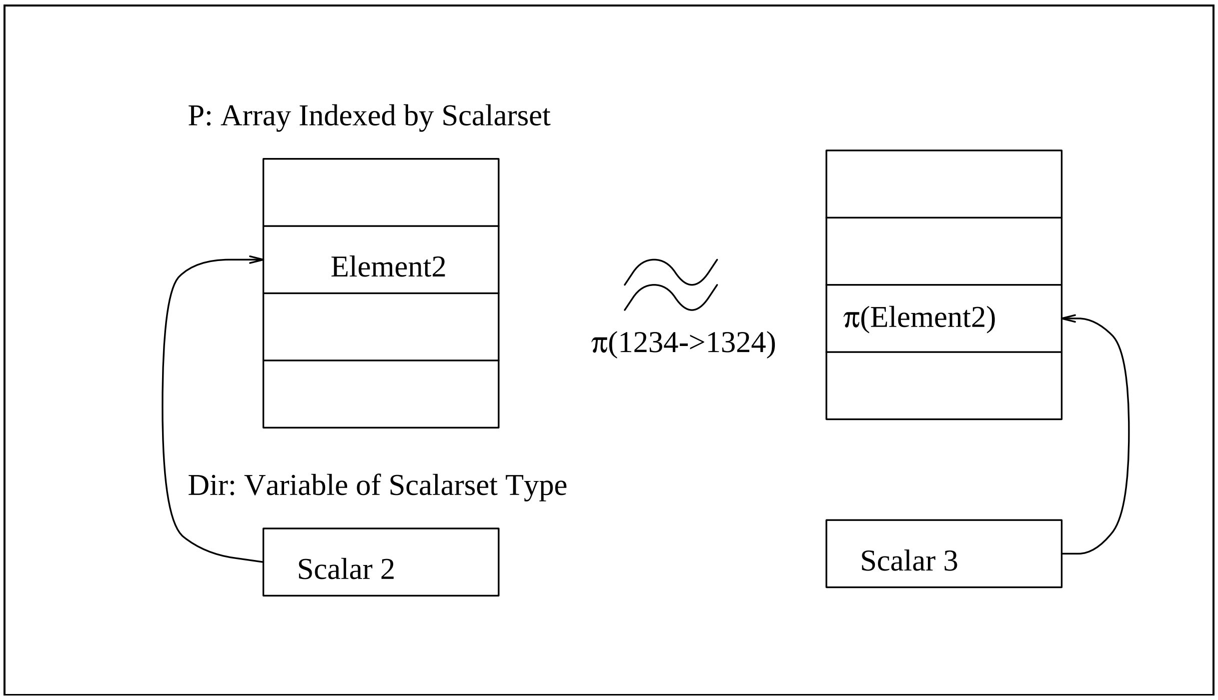


Figure 6: Permutation on State Variable

4) **Exists** *ID: ScalarsetType* **Do** *booleanexpr* **Endexists:** This operation is the conventional $\exists$ operator on a boolean expression.

## 2.4 Construction of Equivalent States

Scalarset values are represented as integers in the states, but we can obtain equivalent states by permuting the scalarset entries of a state. For example, as shown in Figure 6, if we permute the scalarset of size 4 from (1234) to (1324), we first apply the permutation to the individual elements of the array $P$, and then rearrange the positions of the elements. The variable $Dir$ is changed from 2 to 3 accordingly. Therefore, any references to the array $P$ through $Dir$ still gives the corresponding permuted element.

The permutation process can be summarized as follows:

- When the permutation is applied to a scalarset variable, the value is modified to the corresponding permuted value.
- When an array indexed by a scalarset is permuted, the contents of the array elements are permuted and the elements are rearranged according to the permutation.

Note that when we refer to "applying a permutation to a state," we are referring to a one-to-one mapping on the elements of a scalarset, not necessarily a permutation of the state variables.

# 3 Better Verification

We present the properties induced by scalarset declarations in a Mur$\varphi$ program in this section. The properties enable us to reduce the number of states to be inspected during verification. This is essentially an application to Mur$\varphi$ of previous work on symmetries discussed above.

## 3.1 Graph Automorphism

Our result shows that any set of graph automorphisms that is closed under functional composition can be used to combine abstractly-equivalent states in the state space. Here are the formal definitions of state graphs and automorphisms on them:

**Definition 1** *A state graph is a triple:*

$$A = (Q, S, \Delta)$$

| | | |
|---|---|---|
| *where* | $Q$ | *is the set of states.* |
| | $S$ | *is the set of start states, $S \subset Q$.* |
| | $\Delta$ | *is the set of transition rules $r_i : Q \rightarrow Q \cup \{Error\}$.* $\square$ |

**Definition 2** *A graph automorphism on the state graph $A : (Q, S, \Delta)$ is a one-to-one mapping $h : A \rightarrow A$ with the following properties:*

1. *$h$ is a tuple $(h_Q, h_\Delta)$, with $h_Q$ and $h_\Delta$ as bijections:*

$$h_Q : Q \cup \{Error\} \rightarrow Q \cup \{Error\}$$
$$h_\Delta : \Delta \rightarrow \Delta$$

2. *The mapping is closed over the components of $A$:*

$$\forall q \in Q : h_Q(q) \in Q$$
$$\forall q \in S : h_Q(q) \in S$$
$$\forall r \in \Delta : h_\Delta(r) \in \Delta$$
$$h_Q(Error) = Error$$

3. *The transition relation is preserved:*

$$\forall q \in Q, r \in \Delta : h_Q(r(q)) = h_\Delta(r)[h_Q(q)] \qquad \square$$

Any set of graph automorphisms that is closed under functional composition induces an equivalence relation on states:

**Definition 3** *Given a set of graph automorphisms $H$ that is closed under functional composition, two states $s$ and $s'$ are defined to be $H$-equivalent, $s \approx_H s'$, if $s = s'$ or there exists an automorphism $h \in H$ such that $s = h(s')$.*
$\square$

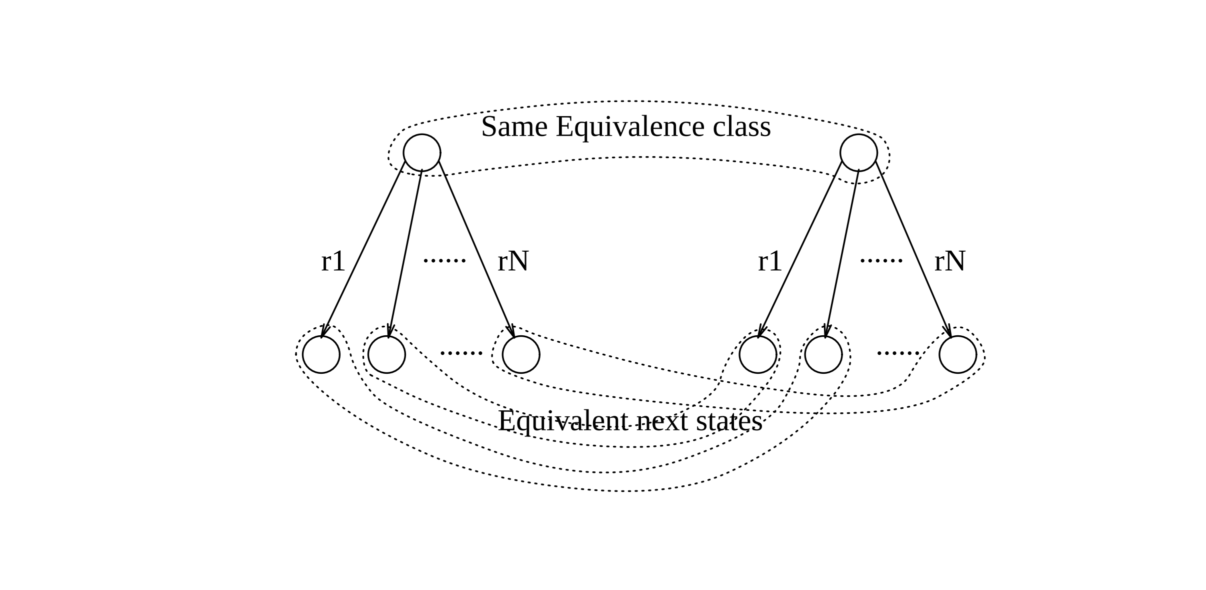


Figure 7: Bisimulation Property of Equivalence Class

Moreover, this equivalence relation is a bisimulation (Figure 7):

> **Theorem 1** *The automorphism equivalence induced by a set of automorphisms $H$ is a bisimulation on the state graph $A : (Q, S, \Delta)$:*
>
> $$\begin{aligned}&\forall p_1, p_2 \in Q : \\ &\quad (p_1 \approx_H p_2 \Rightarrow \forall r_1 \in \Delta : \exists r_2 \in \Delta : r_1(p_1) \approx_H r_2(p_2)) \qquad \Box\end{aligned}$$

Bisimularity is often defined for systems with labelled transitions. The definition here is the obvious analog when states are labelled instead of transitions. In particular, the set of permutations of a scalarset (applied to a state as described above) form a closed set of graph automorphisms.

> **Theorem 2** *The set of permutations $\pi$ on the scalarset entries in the states forms a set of graph automorphisms over the state graph. The set is closed under functional composition and the corresponding equivalence relation $\approx$ is a bisimulation.* $\Box$

## 3.2 Reduced State Space Generation

Because of the properties of automorphism and bisimulation discussed in Section 3.1, the equivalence class obtained from structural symmetry has the following properties:

```
Reached = Unexpanded = { ζ(q) | q ∈StartState }
While Unexpanded ≠ φ Do
   Remove a state s from Unexpanded
     For each enabled rule r from the rule set Do
       Get next state q by firing r
       If ζ(q) is not in Reached
           Put ζ(q) in Reached
           Put ζ(q) in Unexpanded
       Endif
   Endforloop
Endwhileloop
```

Figure 8: On-the-fly Algorithm to Construct the Reduced State Graph

**Theorem 3** *For a verification state graph $A : (Q, S, \Delta)$ and the corresponding equivalence class $[q]$ for any state $q$:*

- *all members of $[q]$ have the same invariant checking results*

$$\begin{aligned}&\forall q \in Q : \forall q_1, q_2 \in [q] : \\ &\quad (\exists r_1 \in \Delta : r_1(q_1) = Error) \Rightarrow (\exists r_2 \in \Delta : r_2(q_2) = Error)\end{aligned}$$

- *the sets of next states of all members of $[q]$ are equivalent.*

$$\begin{array}{ll} let & \Delta(q) = \{r(q) | r \in \Delta\} \\ and & [\Theta] = \{[q] | q \in \Theta\} \text{ where } \Theta \text{ is any set of states.} \\ then & \\ & \forall q \in Q : \forall q_1, q_2 \in [q] : [\Delta(q_1)] = [\Delta(q_2)] \end{array} \qquad \square$$

Mur$\varphi$ allows the user to provide a set of invariants, which are Boolean expressions, at the end of a description. An error is reported whenever the invariant is violated. Here, we are treating the invariants as rules that lead to the "Error" state (so the invariants are assumed to be symmetric). So this theorem implies that if one state of an equivalence class violates the invariant, all of them do. Similarly, if one state is a deadlock (has no successors other than itself), so is every other state in the equivalence class). Currently, Mur$\varphi$ only checks for deadlocks and violations of invariants. However, this result extends easily to the checking of more sophisticated specifications.

With theorem 3, we need search only the (much smaller) quotient graph under the equivalence relation $\approx$, instead of the original graph. We have modified the search algorithm used in the basic Mur$\varphi$ Verification System to generate the quotient of the state graph, as shown in Figure 8. The only change to the algorithm is the introduction of the function $\zeta(q)$, which maps the state $q$ to a unique state representing its equivalence class $[q]$.

The new search algorithm enables us to check equivalent states on-the-fly. Each state is checked as generated; an error is reported as soon as it is found, without generating the whole state space. Using this extended algorithm, we are able to obtain the reduced state graph without generating the original state graph. The maximum reduction is determined by the average size of the equivalence classes.

The following theorem shows that it is possible to check for deadlocks and violations of invariants on the reduced graph:

**Theorem 4** .

- *For every $q$ in the set of unreduced reachable states, $\zeta(q)$ is in the set of reduced reachable states obtained by the algorithm.*
- *An error state $q$ exists in the original state graph if and only if a corresponding error state $[q]$ exists in the reduced state graph.*
- *A deadlock state $q$ exists in the original state graph if and only if a corresponding deadlock state $[q]$ exists in the reduced state and either $\zeta(q)$ has no next states or the only next state (before canonicalization) is $\zeta(q)$ itself.*

## 3.3 Representative of the Symmetry Equivalence Class

As described in Section 3.2, the only change to the verifier is the introduction of the *canonicalization* function $\zeta$. This function determines a unique state $\zeta(q)$ to represent the corresponding equivalence class. Clarke and McMillan [CM89] have observed that finding the canonical state is at least as hard as testing for graph isomorphism, for which no polynomial-time algorithm is known.

Although the problem is inherently exponential, the large reduction in the size of the state spaces compensates for the computation load in canonicalization. And for many practical systems, time reduction in the overall verification process is obtained.

In case of systems with complicated state structure, the computation load in canonicalization may be very high. Fortunately, the following observation allows us to reduce the computation load.

**Observation** *Any subset of states in the equivalence class can be used to represent the class and still give sound verification algorithm for safety properties.*
□

Therefore, *normalization* algorithms can be used to find a subset of states to represent the equivalence class. In Section 4, the examples show that a small subset achieves most of the reduction from a full canonicalization function much more quickly.

We have implemented two algorithms:

**Canonicalization Algorithm:** All permutations are generated and the lexicographically smallest state is used as the canonical state.

**Normalization Algorithm:** We separate the state into two parts. The part with the most significant bits is canonicalized. Because the same lexicographical value may be obtained from different permutations, we may have a few canonicalizing permutations for this part of the state. The second part is normalized by one of the permutations used to canonicalize the first part. The result is a normalized state of a small lexicographically value.

# 4 Practical Results

The new symmetry-based search algorithm has been implemented in the Mur$\varphi$ Verifier System. A wide range of examples were modeled in the new system [ID93]. We present in this section the results from a directory-based cache coherence protocol that was designed at Stanford.

Through a cache coherence protocol, a shared-memory abstraction can be implemented on top of a message-passing network. A typical configuration consists of processing nodes communicating to memory modules via an interconnection network. Each processing node has its own processors and caches.

Maintaining cache coherence is a very complicated task. For example, while many processing nodes have shared copies of some data, another processing node may want to update the data. All shared copies must be invalidated before the data can be updated, so that stale values are not read later. The problem becomes more complicated when many transactions can be initiated at the same time and the messages can be delayed or reordered in the network. A protocol verifier methodically explores all of these possibilities.

The DASH multiprocessor, built at Stanford University [LLG+90], uses a directory-based protocol to keep the caches consistent. The protocol includes normal cache data access, DMA access, special lock operations and many other operations.

The result presented is obtained from the description on the basic protocol (with basic cache data operations : cache read, cache write and write back). The processor-ids, memory module-ids, addresses and data values are modeled as scalarsets. The examples have a single memory module of one 1-bit data address. We have tried examples with 2, 3 and 4 processing nodes. As shown in the table below, the number of states increases quickly as the number of processing nodes increases. However, the use of symmetries reduces the size significantly.

| #Nodes | Algorithm | size | time | % reduction | max possible reduction |
|---|---|---|---|---|---|
| 2 | Unreduced | 1,694 | 12s | 0% | $1 - \frac{1}{2! \times 2!} = 75\%$ |
| | Canonicalized | 425 | 48s | 75% | |
| | Normalized | 429 | 7s | 75% | |
| 3 | Unreduced | 91,254 | 23min | 0% | $1 - \frac{1}{3! \times 2!} = 92\%$ |
| | Canonicalized | 7,741 | 4.5hr | 92% | |
| | Normalized | 9,002 | 13min | 90% | |
| 4 | Unreduced | exceeded 80Mbytes | | | $1 - \frac{1}{4! \times 2!} = 98\%$ |
| | Canonicalized | exceeded 36hr | | | |
| | Normalized | 206,169 | 36hr | — | |

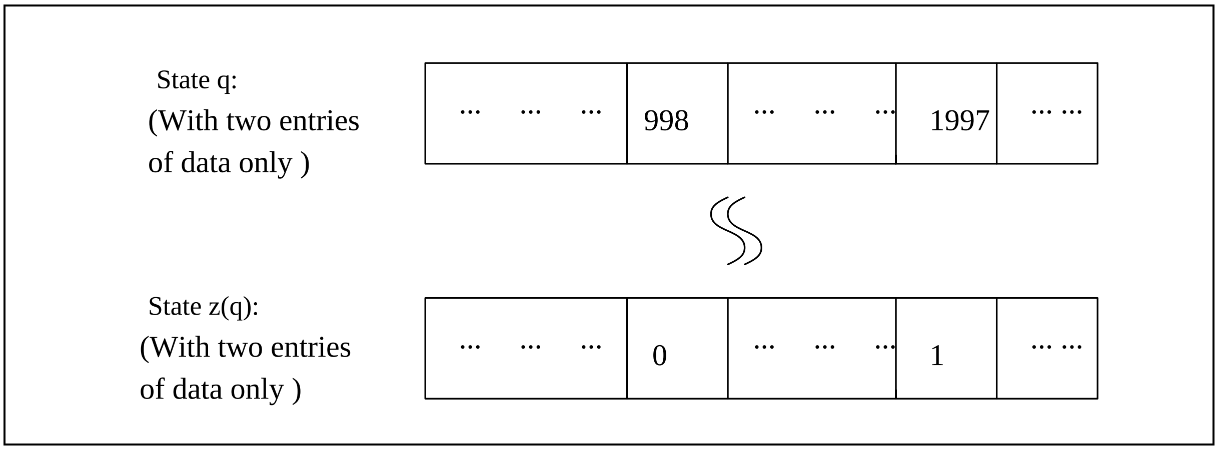


Figure 9: Example on Data Saturation

# 5 Data saturation

In some cases, the use of symmetry can reduce an infinite state space to a finite one. For example, in the cache coherence protocol, the data values can be modeled as a scalarset to obtain a finite reduced state space for a infinite data domain. We obtain a finite reduced state space by the theorem below:

> **Theorem 5** *For any finite system with $M$ scalarsets that are not used as array indexes, there exist finite integers $N_1, ..., N_M$ such that the reduced state graph has the same size as the one obtained from the system with the scalarsets of sizes $N_1, ..., N_M$ or above, even if the sizes approach infinity.* □

The theorem is an immediate corollary of the following lemma:

> **Lemma** *For any finite system with $M$ scalarsets that are not used as array indexes, there are finitely many distinct equivalence classes induced by symmetry represented by the scalarsets, regardless of the sizes of the scalarsets.*
>
> *Furthermore, there exist integers $N_1, ..., N_M$ such that the number of distinct equivalence classes is fixed for scalarsets of size $N_1, ..., N_M$ or above.* □

The main idea behind the lemma is as follows: Let $N$ be the number of distinct values of a given scalarset type that appear in a state $s$, and a map $\zeta$ which maps these values to $0..(N-1)$, independent of the size of the scalarset, we have $s \approx \zeta(s)$. Therefore the number of distinct equivalence class for a scalarset of any size greater than $N$ is the same as the number corresponding to a scalarset of size $N$.

Intuitively, the scalarset that is never used to index an array can be regarded as the data in a data-independence protocol, and the smallest upper bound $N$ corresponds to the maximum number of *distinct* values in a state. For example, in our cache coherence protocol example with one address, most of the states have only one up-to-date value. Sometimes there may be an old value and an up-to-date value. Very rarely there are many old values and an up-to-date value. The upper bound of the number of values for the cache coherent protocol is:

$$N = ((\text{Number of processors}) + 1) \times (\text{Number of address})$$

Using data saturation properties to verify systems with infinite data size is analogous to the approach suggested in Wolper's paper on data independence [Wol86]: An infinite temporal statement can be replaced by a finite statement containing a finite set of data. But Wolper requires the user to recognize that a protocol is data-independent and to transform the description in order to exploit the data-independence. Our method requires no hand transformation (or thought) other than the appropriate use of scalarsets.

# 6 Conclusion

It should be clear by now that, with sufficient effort, any correct design can be verified. It is time to address explicitly the economics of verification, particularly the labor requirements. Our primary contribution here is to make exploitation of structural symmetries *easy,* through simple changes in our description language.

There are many interesting possibilities for further exploration. The same approach can be extended to include other type of symmetries such as rotational symmetry (although the reductions in the state space will not be as dramatic). The ideas presented here can be applied easily to other high-level description languages.

Note that additional analysis could check automatically that a particular subrange is used only in ways that would be valid for a scalarset. This would allow us to have the same benefits of reduction via symmetry without adding the scalarset type to a verification system. Therefore, the symmetry-based technique can be used for other description languages, such as VHDL. We prefer to have an explicit scalarset type in Mur$\varphi$ because the compiler can then report to the user when a symmetry-breaking operation has been unintentionally applied. Also, the more abstract description allowed by using scalarsets may have other advantages; for example, a scalarset can be refined into many different implementations.

The idea of using symmetry in verification can be easily generalized to other specifications and models. Although the algorithm shown is for the verification of asychronous concurrent system, the concept of symmetry can be equally applied to process algebra, synchronous models, etc.

We believe that this is only one of many cases where verification concerns, especially the state explosion problem, can be addressed in part by description language design. Every new purpose for description languages has implications for future languages; we have only begun to explore the implications of automatic verification methods.

# Acknowledgements

We would like to thank Ed Clarke and Allen Emerson for sharing not-yet-published results cited above.

## References


[BWB82] J. Billing, M. C. Wilbur-Ham, and M. Y. Bearman. Automated Protocol Verification. In M. Diaz, editor, *Protocol Specification, Testing, and Verification, V*, pp. 59-70, 1986.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design — a Foundation*, Addison-Wesley, 1988.

[CM89] E.M. Clarke and K.L. McMillan. *Personal Communication,* 1989.

[DDHY92] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, 1992.

[Hol87] Gerard J. Holzmann. *Automated Protocol Validation in Argos, Assertion Proving and Scatter Searching.* Computer Science Press, pp. 163-188, 1987.

[HJJJ84] Peter Huber, Ame M. Jensen, Leif O. Jepsen, and Kurt Jensen. Towards Reachability Trees for High-level Petri Nets. In G. Rozenberg, editor, *Advances on Petri Nets '84*, Springer Verlag, pp. 215–233, 1984.

[ID93] C. Norris Ip and David L. Dill. Efficient Verification of Symmetric Concurrent Systems. To appear in *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Cambridge, MA, October 3-6, 1993.

[LLG+90] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. *Proceedings of 17th International Symposium on Computer Architecture*, May, 1990.

[Sta91] P.H. Starke. Reachability Analysis of Petri Nets Using Symmetries. *Systems Analysis - Modelling - Simululation*, Vol 8, No. 4/5, pp. 293-303, 1991.

[Wol86] P. Wolper. Expressing Interesting Properties of Programs. *13th Annual ACM Symposium on Principles of Programming Languages*, pp. 184-93, 1986.

[ZWR+80] Pitro Zafiropulo, Colin H. West, Harry Rudin, D.D. Cowan, and Daniel Brand. Towards analyzing and synthesizing protocols. *IEEE Transactions on Communications*, COM-28(4), April 1980.