

# Symmetry in Temporal Logic Model Checking

ALICE MILLER, ALASTAIR DONALDSON, AND MUFFY CALDER

*University of Glasgow*

Temporal logic model checking involves checking the state-space of a model of a system to determine whether errors can occur in the system. Often this involves checking symmetrically equivalent areas of the state-space. The use of symmetry reduction to increase the efficiency of model checking has inspired a wealth of activity in the area of model checking research. We provide a survey of the associated literature.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software Program Verification—*Correctness proofs; validation*

General Terms: Theory, Verification

Additional Key Words and Phrases: Model checking, symmetry, quotient graph

## 1. INTRODUCTION

As software-controlled systems expand and become more complex, the importance of error detection at design-time increases. It is estimated [Schneider 2003] that 70% of design-time is spent on simulation to minimize the risk that errors are exposed at a later stage in production, involving costly redesign.

Temporal logic model checking [Clarke et al. 1999; Merz 2000; Müller-Olm et al. 1999] is a technique whereby the properties of a system can be checked by building a model of the system and checking whether the model satisfies these properties. The model is constructed using a specification language, and checked using an automatic model checker. Failure of the model to satisfy a desired property of the system indicates either that the model does not accurately reflect the behavior of the system, or that there is an error (bug) in the original system. Examination of counterexamples provided by the model checker enables the user to either refine the model, or more importantly, to debug the original system.

Any search technique used in model checking involves the exploration of the state-space associated with the model. Inherent symmetry of the original system will be reflected in the state-space. Therefore, knowledge of the symmetry of the system can be used to avoid searching areas of the state-space which are symmetrically equivalent to areas that have been searched previously.

Several different approaches and techniques have been proposed for using symmetry to reduce the size of the state-space to be explored. Some of these have been

---

Authors' addresses: A. Miller, A. Donaldson, M. Calder, Department of Computing Science, University of Glasgow, 17 Lilybank Gardens, Glasgow, UK, G12 8QQ; email: [alice@dcs.gla.ac.uk](mailto:alice@dcs.gla.ac.uk);

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 direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. 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, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or [permissions@acm.org](mailto:permissions@acm.org).

©2006 ACM 0360-0300/2006/09-ART8 \$5.00. DOI 10.1145/1132960.1132962 <http://doi.acm.org/10.1145/1132960.1132962>.

implemented within widely used model checkers [Bosnacki et al. 2002; Hendriks et al. 2003; Ip and Dill 1996; McMillan 1993; Wang and Schmidt 2002]. Indeed, there is even a model checker designed primarily for the verification of highly symmetric systems [Sistla et al. 2000]. In this article we survey and classify the existing research in this area.

Note that some of the methods discussed in this article have recently been surveyed in some detail [Sistla 2004]. Specifically, that article describes methods based on annotated quotient structures (AQSs) and related guarded quotient structures (GQSs). We discuss these approaches in Sections 4.2, 4.6, and 4.7, and describe how they are implemented using the SMC model checker in Section 5.1. Our survey has a much broader scope than that of Sistla [2004]. We include the preceding methods within the wider context of symmetry reduction methods for model checking in general, and we describe a greater range of systems that employ symmetry reduction methods.

## 2. MODEL CHECKING

Temporal logic model checking [Clarke et al. 1986, 1999; Merz 2000; Müller-Olm et al. 1999] is an automatic technique for verifying finite state concurrent systems. The type of systems that we are concerned with maintain a continuing interaction with their environment, and are referred to as *reactive* systems. Model checking involves the construction of a model of a system, usually in the form of a *Kripke structure*, and the verification of temporal logic properties of this model.

Let  $V$  denote a set of variables, and for each  $v \in V$ , let  $D(v)$  be the domain of  $v$ . The set of atomic propositions over  $V$  is given by

$$AP = \{(v = val) : v \in V \text{ and } val \in D(v)\}.$$

For a given set of variables  $V$ , a Kripke structure is defined in terms of  $AP$  thusly:

*Definition 1.* A Kripke structure  $\mathcal{M}$  over  $AP$  is a tuple  $\mathcal{M} = (S, R, L, S_0)$  where:

- (1)  $S$  is a nonempty, finite set of states
- (2)  $R \subseteq S \times S$  is a *total* transition relation that is for each  $s \in S \exists t \in S$  such that  $(s, t) \in R$
- (3)  $L : S \rightarrow 2^{AP}$  is a mapping that labels each state in  $S$  with the set of atomic propositions true in that state
- (4)  $S_0 \subseteq S$  is a set of initial states.

A path in  $\mathcal{M}$  is an infinite sequence of states  $\pi = s_0, s_1, s_2, \dots$  such that  $s_0 \in S_0$ , and for all  $i > 0$ ,  $(s_{i-1}, s_i) \in R$ . Similarly, a transition sequence is an infinite sequence of transitions. For states  $s$  and  $t$ , it is common to denote the transition from  $s$  to  $t$ , by  $s \rightarrow t$ .

We often refer to the *state-space* associated with a system. By this we mean the Kripke structure of the associated transition system. Because of the graphical nature of the state-space, it is sometimes referred to as the *state graph* associated with the system.

In practice, reactive systems are described using modeling languages, including (pseudo) programming languages such as PROMELA [Holzmann 2003] or the SMV language [McMillan 1993], Petri nets [Girault and Valk 2003], or process algebras such as PBC (Petri box calculus) [Best and Koutny 1995] or LOTOS (language of temporal ordering specification) [Bolognesi and Brinksma 1987].

Model checking involves checking the truth of a set of specifications defined using a temporal logic. Generally, the temporal logic that is used is either  $CTL^*$  or one of its

sublogics, *CTL* (computation tree logic) [Clarke et al. 1986] or *LTL* (linear temporal logic) [Pnuelli 1981].

### 2.1. Syntax and Semantics of *CTL\**, *CTL*, and *LTL*

The logic *CTL\** is defined as a set of state formulas, where the *CTL\** state and path formulas are defined inductively to follow. The quantifiers *A* and *E* are used to denote *for all paths*, and *for some path*, respectively (where  $E\phi = \neg A\neg\phi$ ). In addition, *X*, *U*,  $\langle \rangle$ , and  $\square$  represent the standard *nexttime*, *strong until*, *eventually*, and *always* operators (where  $\langle \rangle\phi = \text{true}U\phi$  and  $\square\phi = \neg\langle \rangle\neg\phi$ , respectively). Let *AP* be a finite set of propositions. Then

- for all  $p \in AP$ ,  $p$  is a state formula
- if  $\phi$  and  $\psi$  are state formulas, then so are  $\neg\phi$ ,  $\phi \wedge \psi$ , and  $\phi \vee \psi$
- if  $\phi$  is a path formula, then  $A\phi$  and  $E\phi$  are state formulas
- any state formula  $\phi$  is also a path formula
- if  $\phi$  and  $\psi$  are path formulas, then so are  $\neg\phi$ ,  $\phi \wedge \psi$  and  $\phi \vee \psi$ ,  $X\phi$ ,  $\phi U\psi$ ,  $\langle \rangle\phi$ , and  $\square\square\phi$ .

The logic *CTL* is the sublogic of *CTL\** in which the temporal operators *X*, *U*,  $\langle \rangle$ , and  $\square$  must be immediately preceded by a path quantifier. The logic *LTL* is obtained by restricting the set of (*CTL\**) formulas to those of the form  $A\phi$ , where  $\phi$  does not contain *A* or *E*. When referring to an *LTL* formula, it is conventional to omit the *A* operator and instead, interpret the formula  $\phi$  as “for all paths  $\phi$ .”

For a model  $\mathcal{M}$ , if the *CTL\** formula  $\phi$  holds at a state  $s \in S$ , then we write  $\mathcal{M}, s \models \phi$  (or simply  $s \models \phi$  when the identity of the model is clear from the context). The relation  $\models$  is defined inductively to follow. Note that for a path  $\pi = s_0, s_1, \dots$ , starting at  $s_0$ ,  $\text{first}(\pi) = s_0$ , and for all  $i \geq 0$ ,  $\pi_i$  is the suffix of  $\pi$  starting from state  $s_i$ .

- $s \models p$ , for  $p \in AP$  if and only if  $p \in L(s)$
- $s \models \neg\phi$  if and only if  $s \not\models \phi$
- $s \models \phi \wedge \psi$  if and only if  $s \models \phi$  and  $s \models \psi$
- $s \models \phi \vee \psi$  if and only if  $s \models \phi$  or  $s \models \psi$
- $s \models A\phi$  if and only if  $\pi \models \phi$  for every path  $\pi$  starting at  $s$
- $\pi \models \phi$ , for any state formula  $\phi$ , if and only if  $\text{first}(\pi) \models \phi$
- $\pi \models \neg\phi$  if and only if  $\pi \not\models \phi$
- $\pi \models \phi \wedge \psi$  if and only if  $\pi \models \phi$  and  $\pi \models \psi$
- $\pi \models \phi \vee \psi$  if and only if  $\pi \models \phi$  or  $\pi \models \psi$
- $\pi \models \phi U\psi$  if and only if, for some  $i \geq 0$ ,  $\pi_i \models \psi$  and  $\pi_j \models \phi$  for all  $0 \leq j < i$
- $\pi \models X\phi$  if and only if  $\pi_1 \models \phi$
- $\pi \models \langle \rangle\phi$  if and only if  $\pi_i \models \phi$ , for some  $i \geq 0$
- $\pi \models \square\phi$  if and only if  $\pi_i \models \phi$ , for all  $i \geq 0$ .

### 2.2. Büchi Automata and *LTL*

One of the most efficient algorithms for model checking *LTL* properties is the automata-theoretic approach (see Section 2.3.2). Although we will not describe the algorithms in detail, we provide a little background theory here.

*Definition 2.* A *finite-state automaton* (FSA)  $\mathcal{A}$  is a tuple  $\mathcal{A} = (S, s_0, L, T, F)$  where:

- (1)  $S$  is a nonempty, finite set of states
- (2)  $s_0 \in S$  is an initial state

- (3)  $L$  is a finite set of labels
- (4)  $T \subseteq S \times L \times S$  is a set of transitions, and
- (5)  $F \subseteq S$  is a set of final states.

A run of  $\mathcal{A}$  is an ordered, possibly infinite, sequence of transitions

$$(s_0, l_0, s_1), (s_1, l_1, s_2), \dots,$$

where  $s_i \in S$  and  $l_i \in L$  for all  $i > 0$ . An accepting run of  $\mathcal{A}$  is a finite run in which the final transition  $(s_{n-1}, l_{n-1}, s_n)$  has the property that  $s_n \in F$ .

In order to reason about infinite runs of an automaton, alternative notions of acceptance, for example, Büchi acceptance, are required. We say that an infinite run (of an FSA) is an accepting  $\omega$ -run (i.e., it satisfies Büchi acceptance) if and only if some state in  $F$  is visited infinitely often in the run. A Büchi automaton is an FSA defined over infinite runs (together with the associated notion of Büchi acceptance).

Every *LTL* formula can be represented as a Büchi automaton (see, for example, Wolper et al. [1983] and Vardi and Wolper [1994], and references therein). For more details of automata and logic, see, for example, Holzmann [2003].

### 2.3. Model Checking Algorithms

The model checking problem can be stated as follows:

Given a Kripke structure  $\mathcal{M}$  and a temporal logic formula  $\phi$ , determine the set of initial states in  $\mathcal{M}$  that satisfy  $\phi$ .

Generally, we say that the model  $\mathcal{M}$  satisfies the specification  $\phi$  if all of the initial states of  $\mathcal{M}$  satisfy  $\phi$ .

In this section we give a brief overview of the basic model checking algorithms for checking *CTL* and *LTL* formulas, respectively, and describe the techniques that are used to combat the state-space explosion problem for each. Note that *CTL*<sup>\*</sup> model checking was first introduced in Clarke et al. [1986]. A method for checking *CTL*<sup>\*</sup> properties [Emerson and Lei 1987] involves the use of an *LTL* model checker on the subformulas of the property to be checked. However, most model checkers are used to verify either *CTL* or *LTL* properties, but not both.

*2.3.1. CTL Model Checking.* The model checking algorithm for *CTL* [Clarke et al. 1986, 1994; Quielle and Sifakis 1982] works by successively marking the states which satisfy subformulas of the formula to be checked. The particular form of algorithm used depends on the formula. For illustration, we give here an example of how the algorithm proceeds to check formula  $\phi$ , where  $\phi$  is  $A(\phi_1 \cup \phi_2)$ .

For a state  $s$ ,  $s \models \phi$  if and only if either  $s$  satisfies  $\phi_2$  or  $s$  has at least one successor,  $s$  satisfies  $\phi_1$ , and all successors of  $s$  satisfy  $\phi$ . Initially, all states are marked to indicate whether they satisfy  $\phi_1$  and/or  $\phi_2$  and/or  $\phi$ , and also with a number (*nb* say) denoting how many successors have yet to be marked as satisfying  $\phi$ . Initially, for each state  $s$ , *nb* is set to 0 if  $s \models \phi$ , or to the number of successors of  $s$ , otherwise. In the latter case, each time a successor of  $s$  is marked as satisfying  $\phi$ , *nb* is decremented by one. When *nb* = 0 for  $s$ , clearly  $s \models \phi$ . When no states can be remarked, the algorithm terminates. If at this point all initial states are marked as satisfying  $\phi$ , then  $\mathcal{M} \models \phi$ .

*2.3.2. LTL Model Checking.* The model checking problem for *LTL* can be restated as: “given  $\mathcal{M}$  and  $\phi$ , does there exist a path of  $\mathcal{M}$  that does not satisfy  $\phi$ ?” One approach to *LTL* model checking is the tableau approach described in Müller-Olm et al. [1999].

However, we concentrate here on the more efficient automata-theoretic approach [Lichtenstein and Pnueli 1985; Vardi and Wolper 1986].

In order to verify an *LTL* property  $\phi$ , a model checker must show that all paths of a model  $\mathcal{M}$  satisfy  $\phi$  (alternatively, must find a counterexample, namely, a path which does *not* satisfy  $\phi$ ). To do this, an automaton  $\mathcal{A}$  representing the reachable states of  $\mathcal{M}$  is constructed, together with an automaton  $\mathcal{B}_{\neg\phi}$ , which accepts all paths for which  $\neg\phi$  holds. The asynchronous product of the two automata,  $A'$ , is constructed (in practice,  $A'$  is usually constructed implicitly by letting  $A$  and  $\mathcal{B}_{\neg\phi}$  take alternate steps). Any accepting run of  $A'$  signifies an error. If there are no accepting runs,  $\mathcal{M} \models \phi$ . Generally, to prove *LTL* properties a depth-first search is used. As the search progresses, all states visited are stored (in a reduced form) in a hash array (or heap), and states along the current path are pushed on to the stack.

If the property  $\phi$  to be verified is a safety property, say  $\phi = \Box\psi$ , where  $\psi$  does not contain the until operator  $U$ , then a depth-first search of  $A'$  is used. If a state is encountered at which  $\psi$  is false, then  $\phi$  is false and the current path (the current contents of the stack) provides a counterexample. If, on the other hand,  $\phi$  is a liveness property, then determining the truth, or otherwise, of  $\phi$  relies on the ability to detect the presence of infinite accepting runs in  $A'$ . This is achieved either by using the classic approach of Tarjan [1972], in which strongly connected components are constructed and analyzed separately for acceptance runs or via a nested depth-first search [Courcoubetis et al. 1992]. The latter is more efficient than the classic approach in that it is not necessary to produce all acceptance runs, but merely a single acceptance cycle (if one exists). Suppose, for example,  $\phi$  is  $\Box(\Diamond)p$  for some proposition  $p$ . From any state  $s$  reached during an initial search at which  $\neg p$  holds, a second search is initiated to check for paths leading back to  $s$ , during which  $p$  remains false. If no such path exists, the original search resumes from  $s$ .

*2.3.3. The State-Space Explosion Problem.* One of the main problems associated with model checking is that of *state-space explosion*. The model checking algorithms described previously both rely upon the explicit construction of a model representing all possible system states. The number of states for a system with associated model  $\mathcal{M}$  is potentially exponential with respect to the number of processes in the system. As a result, full verification is often impossible. Therefore, techniques are used to try to reduce the memory required to store each state (e.g., symbolic state representation, see the following) or the number of states or paths explored (e.g., on-the-fly methods and partial order reduction, see the following). Another method used to reduce the number of states is symmetry reduction, in which subsets of *symmetrically equivalent* states are collapsed into a single *representative* state. Symmetry reduction is described in full in Section 4.

*2.3.4. Symbolic Model Checking.* Symbolic model checking [Burch et al. 1992] is a method by which the states and transitions of a model are represented *symbolically* (as opposed to *explicitly*) in order to save space. A particular symbolic approach (namely, *BDD*-based encoding) has proved especially successful for the verification of *CTL* properties for very large systems [McMillan 1993].

A binary decision tree is a structure that is used to represent a Boolean formula. Any assignment of truth values to the variables of the formula corresponds to a path down the tree from the root node to a terminal node, which is labeled either true or false. The value of this label determines the value of the function for this assignment of variables. A binary decision diagram (*BDD*) is obtained from a binary decision tree by merging isomorphic subtrees and identical terminals. Any set of states can be encoded

as a *BDD*. Indeed, if  $S$  is a set of states encoded as a set of Boolean tuples (on a set  $X$ ), then for any fixed ordering of the elements of  $X$ , there is a unique *BDD* representing  $S$  [Bryant 1992].

An ordered binary decision diagram (*OBDD*) is a *BDD* which has a total ordering applied to the variables labeling the vertices of the diagram. The size of the *OBDD* can vary greatly, depending on the ordering used. Heuristics have been developed to find efficient orderings for a given formula (when such an ordering exists). However, finding the optimal ordering is NP-complete [Bollig and Wegener 1996].

For a Kripke structure, both the set of states and set of transitions can be represented by *BDDs*. All *possible* states are encoded, as opposed to all *reachable* states. As superfluous states are unreachable, they do not affect the result of model checking. Indeed, their presence may lead to a simplification of certain *BDDs*. In addition, it is possible to first compute the reachable states,  $R$  say, and then restrict the CTL model checking algorithm to  $R$ .

*2.3.5. On-the-Fly Model Checking.* It is not always necessary to build the entire state-space in order to determine whether or not a system satisfies a given property.

If the property to be checked is false, only part of the state-space needs to be constructed; up to the point at which an error state (safety property) or violating cycle (liveness property) is discovered. However, if there are no errors, the entire state-space must be constructed. This means that although debugging can be performed relatively easily, property verification very quickly becomes prohibitive.

On-the-fly methods are most suitable for model checking algorithms based on a depth-first traversal of the state-space (i.e., explicit state methods) and have been developed to check specifications in *LTL*, *CTL*, and *CTL\** [Vardi and Wolper 1986; Vergauwen and Lewi 1993; Bhat et al. 1995].

Some approaches for combining on-the-fly techniques with symbolic model checking exist [Ben-David and Heyman 2000], but are restricted to checking safety properties.

*2.3.6. Partial Order Reduction.* The explosion of states and transitions in a model results from the interleaving of actions of distinct processes in all possible orders. In general, the consideration of all such interleavings is crucial—bugs in concurrent systems often correspond to the unexpected ordering of actions. However, if a set of transitions is entirely independent and *invisible* with respect to the property being verified, the order in which the transitions are executed does not affect the overall behavior of the system (a transition is invisible with respect to a property  $\phi$  if the truth of  $\phi$  is unaffected by the transition). Partial order reduction [Emerson et al. 1997; Godefroid 1996b; Peled 1996a] exploits this fact, and considers only one representative ordering for any set of concurrently enabled, independent, invisible transitions.

Partial order reduction methods rely on determining a suitable subset of transitions to be considered at every state. As a result, rather than exploring a structure  $\mathcal{M}$ , an equivalent (usually smaller) structure  $\mathcal{M}'$  is explored, with fewer transitions and states.

The particular subset (and correspondingly, equivalence relation) depends upon the strategy being used. A common strategy, for example, is the *ample sets* method [Peled 1996b]. This is the method chosen for the partial order reduction implementation in SPIN [Holzmann 2003; Holzmann and Peled 1994]. In this case, suppose that a property  $\phi$  is to be verified. For any state  $s$  reached along a search path, rather than considering all of the transitions enabled at  $s$  (*enabled(s)*), an ample set (*ample(s)*) of transitions is chosen in such a way to ensure that

—any transition  $t \in \text{enabled}(s)$  which is not in *ample(s)* is independent of all transitions in *ample(s)*. That is, the execution of  $t$  does not affect the enabledness of any of the transitions in *ample(s)*, and vice versa.

- All transitions in  $ample(s)$  are invisible.
- If  $a \in ample(s)$ , then the state resulting from taking transition  $a$  from  $s$  has not been reached along the current search path.

The equivalence relation in this case is *trace* equivalence. Two transition sequences are said to be trace equivalent if one can be obtained from the other by repeatedly commuting the order of adjacent, independent transitions. Using the ample sets method, every transition sequence in the original structure  $\mathcal{M}$  is trace equivalent to a transition sequence in the reduced structure  $\mathcal{M}'$ . It follows that for any stuttering-closed [Peled 1996b] *LTL* formula  $\phi$ ,  $\phi \models \mathcal{M}$  if and only if  $\phi \models \mathcal{M}'$ .

Other strategies for determining suitable subsets of transitions include the stubborn sets method [Valmari 1992] or the sleep sets and persistent sets method [Godefroid 1996b] which is implemented in VeriSoft [Godefroid 1997].

For some systems wherein all actions are interdependent, partial order reduction cannot offer any improvement in verification space or time. In many realistic cases, however, partial order reduction can be extremely effective. For example, for some systems the growth of the state-space as the number of processes increases is reduced from exponential to polynomial when partial order methods are used. In others, the global state-space may increase with the growth of a parameter, whereas the size of the reduced state-space remains unchanged [Godefroid 1996a].

## 2.4. Example Model Checkers

*2.4.1. Explicit State-Based Model Checkers.* Two of the most popular on-the-fly, explicit-state-based model checkers are SPIN [Holzmann 2003] and Mur $\phi$  [Dill et al. 1992, 1996].

SPIN is used for efficient software verification. Specifications are described using the high-level state-based description language PROMELA (process meta language), which is loosely based on Dijkstra's guarded command language [Dijkstra 1976]. PROMELA allows for the expression of nondeterminism, asynchronous and synchronous communication, dynamic process creation, and mobile communications (communication channels can contain references to other communication channels). SPIN uses a depth-first search algorithm (breadth-first search is also possible) and can be used as a full *LTL* model checking system supporting all correctness requirements expressible in linear-time temporal logic (or Büchi automata, directly). It can also be used as an efficient on-the-fly verifier for more basic safety and liveness properties (e.g., progress and lack of deadlock), which can often be expressed and verified without the use of *LTL*.

SPIN has been used to trace logical errors in distributed systems designs, such as operating systems [Cattel 1994; Kumar and Li 2002], computer networks [Yuen and Tjioe 2001], and railway signaling systems [Cimatti et al. 1997], and for the feature interaction analysis of telecommunications and email systems [Calder and Miller 2001, 2003; Holzmann and Smith 1999b].

To optimize verification runs, SPIN uses efficient partial order reduction techniques, and also employs *statement merging* [Holzmann 1999], a special case of partial order reduction that merges internal, invisible process statements to reduce the number of reachable system states. For efficient state-storage, SPIN offers state compression (a form of byte-sharing) or, alternatively, BDD-like storage techniques based on minimized automata [Visser and Barringer 1996]. In addition, approximate hashing methods are available, namely, hash-compact methods [Wolper and Leroy 1993] and bitstate hashing [Holzmann 1998].

The Mur $\phi$  description language is based on a collection of guarded commands (condition/action rules), which are executed repeatedly in an infinite loop. The data

structures and guarded commands are written in an imperative-style language, together with new data types, including “multiset” (for describing a bounded set of values whose order is irrelevant to the behavior of the description) and “scalarset” (for describing a subrange whose elements can be freely permuted). The Mur $\phi$  verifier performs a depth- or breadth-first search over the state-space to check for deadlock or *assertion* or *invariance* violations. Assertion violations are trapped using an *assert* statement (a conditional error statement) within the program description. Invariants, on the other hand, are defined in a separate part of the Mur $\phi$  description. More complex temporal properties cannot be verified.

Other state-based verifiers include PROD [Varpaaniemi et al. 1995] and PEP [Best and Grahlmann 1996], in which systems are specified using Petri nets. The on-the-fly verification of various temporal or  $\mu$ -calculus properties of LOTOS specifications is achieved by translation into to state-spaces using CÆSAR [Garavel and Sifakis 1990], which are then checked using the model checkers XTL [Mateescu and Garavel 1998] or EVALUATOR [Mateescu 2003], respectively. The tool COSPAN [Kurshan 1995] uses an automata-theoretic approach. The system to be verified is modeled as a collection of coordinating processes described in the S/R (selection/resolution) modeling language. The verifier supports both on-the-fly enumerative search and symbolic search using BDDs.

*2.4.2. Symbolic Model Checkers.* The most successful (OBDD-based) symbolic model checker is the branching time *CTL* model checker SMV [McMillan 1993]. Systems are described using the SMV language, which has been developed with a precise semantics that relates programs to their expressions as Boolean formulas. SMV supports both synchronous and asynchronous communication, and provides for modular hierarchical descriptions and the definition of reusable components. It has been used to verify various hardware systems, including an avionics triple sensor voter [Danjani-Brown et al. 2003], the Gigamax cache coherence protocol [McMillan and Schwalbe 1992] and the *t9000* virtual channel processor [Barrett 1995]. NuSMV [Cimatti et al. 1999, 2002] is a reimplemented and extended version of SMV. The additional features contained in NuSMV include a textual interaction shell and graphical interface, extended model partitioning techniques, and facilities for *LTL* model checking.

An enhanced version of SMV, RuleBase [Beer et al. 1996] is an industry-oriented tool for the verification of hardware designs. In an effort to make the specification of *CTL* properties easier for the nonexpert, RuleBase supports its own language, Sugar. In addition, RuleBase supports standard hardware description languages such as VHDL and Verilog.

*2.4.3. Real-Time Model Checkers.* When modeling certain critical systems, it is essential to include some notion of time. If time is considered to increase in discrete steps (discrete-time), then existing model checkers can be readily extended [Alur and Henzinger 1992; Emerson 1992]. The most widely used *dense* real-time model checker (in which time is viewed as increasing continuously) is UPPAAL [Larson et al. 1997]. Models are expressed as timed automata [Alur and Dill 1993] and properties defined in UPPAAL logic, a subset of timed computational tree logic (*TCTL*) [Alur et al. 1990]. UPPAAL uses a combination of on-the-fly and symbolic techniques [Larson et al. 1995; Yi et al. 1994] so as to reduce the verification problem to that of manipulating and solving simple constraints. Another real-time model checker is KRONOS [Yovine 1997], which is used to analyse real-time systems modeled in several timed process description formalisms, such as ATP [Nicollin and Sifakis 1994] and ET-LOTOS [Léonard and Leduc 1997, 1998]. A real-time extension to COSPAN [Alur and Kurshan 1995] allows



real-time constraints to be expressed by associating lower and upper bounds on the time spent by a process in a local state. An execution is said to be *timing-consistent* if its steps can be assigned real-valued time-stamps that satisfy all the specified bounds.

The probabilistic model checker PRISM [Kwiatkowska et al. 2002; Rutten et al. 2004] allows time to be considered as increasing either in discrete steps or continuously. Models are expressed in the PRISM Language and converted to a variant of a Markov chain (either discrete- or continuous-time). Properties are written in terms of probabilistic computation tree logic (PCTL) or continuous stochastic logic (CSL), respectively. Models can also be expressed using PEPA (performance evaluation process algebra) [Hillston 1996] and converted to PRISM.

The hybrid model checker HYTECH [Henzinger et al. 1997] is used to analyze dynamical systems whose behavior exhibits both discrete and continuous change. Linear hybrid automata (extensions to timed automata, including access to dynamic variables) are used to incorporate the discrete behavior of computer programs with the continuous behavior of environmental variables, such as time.

*2.4.4. Direct Model Checking of Programs.* Finite state model checking traditionally requires the manual construction of a model, via a modeling language, which is then converted to a Kripke structure (or finite-state automaton) for model checking.

Recently, there has been much interest in applying model checking directly to program source code written in languages such as Java or C. Early approaches to model checking Java software, like JCAT [Demartini et al. 1999] and Java PathFinder (JPF1) [Havelund and Pressburger 2000], involved the direct translation of Java code into Promela, and subsequent verification via SPIN. Although both of these systems were successful, direct translation meant that programs were only able to contain features that were supported by both Java and Promela (this is not true for floating point numbers, for example).

The BANDERA tool [Corbett et al. 2000] avoids direct translation by instead extracting an abstracted finite-state model from Java source code. This model is then translated into a suitable modeling language (Promela or SMV) and model checked accordingly. Meanwhile, a second-generation Java PathFinder tool (JPF2) [Visser et al. 2000], which makes extensive use of BANDERA abstraction tools, has been developed to model check Java bytecode directly.

The dSPIN tool [Iosif and Sisto 1999] is an extension of SPIN, which has been designed for modeling and verifying object-oriented software (Java programs, in particular). In addition to the usual features available with SPIN, the dSPIN model checker allows for the dynamic creation of heap objects and the representation of garbage collection.

The Bogor model checking framework [Robby et al. 2003] is used to check sequential and concurrent programs. The behavioral aspects of the program are first specified in JML (Java modeling language), which, together with the original Java program, is then translated into a lower-level specification for verification. Bogor exploits the canonical heap representation of dSPIN and is implemented as an Eclipse [Clayberg and Rubel 2004] plug-in.

Various tools address the problem of direct model checking of C code. For example, BLAST (Berkeley lazy abstraction software verification tool) [Henzinger et al. 2003] uses an iterative process of abstraction, verification, and counterexample-driven refinement for proving the correctness of software. The FeaVer (feature verification system) tool [Holzmann and Smith 1999a] allows models to be extracted mechanically from the source of software applications, and checked using SPIN. Indeed, a new feature of SPIN is to allow C code to be embedded directly within a PROMELA specification. Microsoft's SDV (static driver verifier) tool uses the SLAM [Ball et al. 2004] analysis engine to analyze

the source code of Windows device drivers. SDV involves a similar abstraction, verification, and refinement loop to that of BLAST and exploits the BEBOP model checker during the verification stage.

The VeriSoft model checker [Godefroid 1997] is used to verify concurrent processes executing C code. Unlike traditional model checking techniques, the use of VeriSoft does not rely on states being expressed as sequences of bits. Systematic search of the state-space allows the user to check for deadlock and assertion violations, as well as for timeouts and livelocks. A *stateless* search is used, whereby only states along the current path are stored, together with as many states as possible in the remaining available memory. As a result, state-space explosion is not a problem—it is theoretically possible to verify systems of any size. However, as a result, the same path may be explored many times, and so the search can be very slow.

Note that we have not attempted to provide an exhaustive description of the available model checkers, but merely to provide sufficient coverage for the purposes of our symmetry survey. We do not, for example, consider conformance checkers or combination checkers. For an overview of these types of checkers, refer to Clarke and Wing [1996].

### 3. BASIC GROUP THEORY

In this section we summarize some definitions from group theory which will be useful throughout our article.

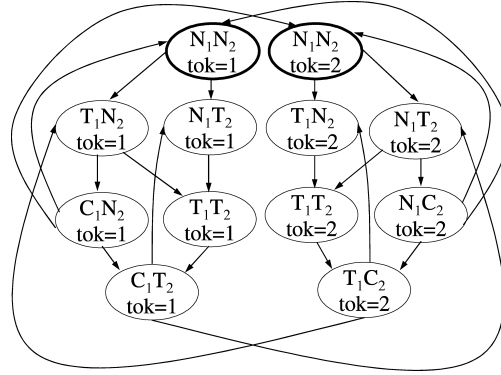
*Definition 3.* Let  $G$  be a nonempty set, and let  $\circ : G \times G \rightarrow G$  be a binary operation. We say that  $(G, \circ)$  is a *group* if:

- $\circ$  is associative
- $G$  has an identity element  $1_G$  and
- for each element  $\alpha \in G$ , there is an inverse element  $\alpha^{-1} \in G$  such that  $\alpha \circ \alpha^{-1} = \alpha^{-1} \circ \alpha = 1_G$ .

We call the operation  $\circ$  *multiplication* in  $G$ . When it is clear what the binary operation is, we simply refer to a group as  $G$ , rather than  $(G, \circ)$ , and use concatenation to denote multiplication. Let  $H$  be a nonempty subset of a group  $G$ . If  $H$  is a group in its own right under the binary operation of  $G$ , that is, it satisfies Definition 3, then we call  $H$  a subgroup of  $G$  and write  $H \leq G$ . For elements  $\alpha_1, \alpha_2, \dots, \alpha_n$  of a group  $G$ , the set  $\langle \alpha_1, \alpha_2, \dots, \alpha_n \rangle$  denotes the smallest subgroup of  $G$  containing  $\alpha_1, \alpha_2, \dots, \alpha_n$ , and is called the subgroup *generated* by  $\alpha_1, \alpha_2, \dots, \alpha_n$ .

Let  $X$  be a finite set. A permutation of  $X$  is a bijection from  $X$  to  $X$ . The set of all permutations of  $X$ ,  $Sym X$ , forms a group under composition of mappings. Any subgroup of this group is called a permutation group acting on the set  $X$ . If  $G$  is a permutation group acting on  $X$  and  $\alpha_1$  and  $\alpha_2$  elements of  $G$ , then for any  $x \in X$ ,  $\alpha_1(x)$  denotes the result of applying  $\alpha_1$  to  $x$ , and  $\alpha_1\alpha_2(x) = \alpha_1(\alpha_2(x))$  the result of applying  $\alpha_1$  to  $\alpha_2(x)$ . Consider the set  $[n] = \{1, 2, \dots, n\}$ . The group of all permutations acting on  $[n]$  is called the *symmetric group on  $n$  points*, and is denoted  $S_n$ , or equivalently,  $S_{\{1, 2, \dots, n\}}$ .

If  $G$  is a permutation group acting on a finite set  $X$ , then for  $x \in X$ , the set  $\{\alpha(x) : \alpha \in G\}$  is called the *orbit* of  $x$  under  $G$ , denoted  $[x]_G$ . For  $x$  and  $y \in X$ , we use  $x \sim_G y$  to denote that  $x$  and  $y$  are in the same orbit. The relation  $\sim_G$  is an equivalence relation on  $X$ , and hence partitions  $X$  into disjoint subsets, which are the orbits of  $X$  under  $G$ . Sometimes it is useful to identify the orbits of a set  $X$  under a group  $G$  as a set of representative elements (one chosen for each orbit). We use the notation  $rep([s]_G)$  to denote the representative element of the orbit containing the element  $s$ .



**Fig. 1.** Kripke structure for two-process mutual exclusion.

## 4. SYMMETRY REDUCTION METHODS IN MODEL CHECKING

### 4.1. Symmetry Reduction in Automatic Verification

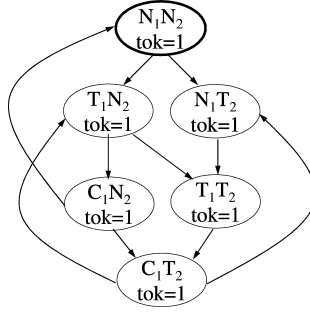
The earliest use of symmetry reduction in automatic verification was in the context of high-level (colored) Petri nets [Huber et al. 1984], where reduction by equivalent markings was used to construct finite reachability trees. These ideas were later extended for deadlock detection and the checking of liveness properties in place/transition nets [Starke 1991].

Concurrent systems often contain many replicated components, and as a consequence, model checking may involve making a redundant search over equivalent areas of the state-space. For example, Figure 1 shows a Kripke structure for a model of two-process mutual exclusion. The model consists of two-processes, each with three local states  $N$ ,  $T$ , and  $C$ . For process  $i$ , the proposition  $N_i$  denotes that process  $i$  is in the *neutral* state. Similarly, the propositions  $T_i$  and  $C_i$  denote that process  $i$  is in the *trying* and *critical* state, respectively. There is a global variable,  $tok$ , which takes the value 1 or 2, depending which process holds the token. Only if process  $i$  is in the trying state (i.e.,  $T_i$  holds) and  $tok = i$  also holds can process  $i$  move into the critical state. When a process leaves the critical state, the token is nondeterministically assigned the value 1 or 2. Thus, in the model it is not possible for both processes to be in the critical state. That is, the mutual exclusion property holds. Note that there are two initial states (indicated with a bold outline in Figure 1). In each initial state, both processes are in the neutral state and one of the processes has the token.

Though simple, this example clearly demonstrates the existence of symmetry within a Kripke structure. In terms of the mutual exclusion property, any pair of states  $(A_1B_2, token = i)$  and  $(B_1A_2, token = j)$ , where  $A$  and  $B$  belong to  $\{N, T, C\}$  and  $i \neq j$ , are *equivalent*, that is, state  $(A_1B_2, token = i)$  will satisfy the mutual exclusion property if and only if  $(B_1A_2, token = j)$  does. Most symmetry reduction techniques exploit this type of symmetry by restricting the state-space search to equivalence class representatives, and often result in significant savings in memory and verification time [Bosnacki et al. 2002; Clarke et al. 1996; Emerson and Sistla 1996; Ip and Dill 1996].

### 4.2. Symmetry Reduction Using Quotient Structures

Let  $\mathcal{M} = (S, R, L, S_0)$  be a Kripke structure over a set of atomic propositions AP. An *automorphism* of  $\mathcal{M}$  is a permutation  $\alpha : S \rightarrow S$  that preserves the transition relation.



**Fig. 2.** Quotient Kripke structure for two-process mutual exclusion.

In other words,  $\alpha$  satisfies the following condition:

$$\forall s, t \in S, (s, t) \in R \Rightarrow (\alpha(s), \alpha(t)) \in R.$$

In a model of a concurrent system with many replicated processes, Kripke structure automorphisms may involve the permutation of process identifiers or data values throughout all states of the model. On the other hand, a model may include a data structure which has geometrical symmetry [Holzmann and Joshi 2004]. In this case, Kripke structure automorphisms involve applying the geometrical symmetries throughout all states of the model.

The set of all automorphisms of the Kripke structure  $\mathcal{M}$  forms a group under composition of mappings. This group is denoted  $Aut(\mathcal{M})$ . A subgroup  $G$  of  $Aut(\mathcal{M})$  partitions the set  $S$  of states into disjoint orbits, as described in Section 3, which can be used to define a *quotient* Kripke structure  $\mathcal{M}_G$ :

*Definition 4.* The quotient Kripke structure  $\mathcal{M}_G$  of  $\mathcal{M}$ , with respect to  $G$ , is a tuple  $\mathcal{M}_G = (S_G, R_G, L_G, S_G^0)$  where:

- $S_G = \{[s]_G : s \in S\}$  (the set of orbits of  $S$  under the action of  $G$ )
- $R_G = \{([s]_G, [t]_G) : (s, t) \in R\}$
- $L_G([s]_G) = L(rep([s]_G))$ , where  $rep([s]_G)$  is a unique representative of  $[s]_G$
- $S_G^0 = \{[s]_G : s \in S_0\}$  (the orbits of the initial states  $S_0$  under the action of  $G$ ).

If  $G$  is nontrivial, then the quotient structure  $\mathcal{M}_G$  is smaller than  $\mathcal{M}$ . For any  $s$ , the size of  $[s]_G$  is bounded by  $|G|$ , and so the theoretical minimum size of  $S_G$  is  $|S|/|G|$ . Since for highly symmetric systems we may have  $|G| = n!$ , where  $n$  is the number of components, symmetry reduction potentially offers a considerable reduction in memory requirements.

In practice, the set of states  $S$  is taken to be the set of orbit representatives, rather than the set of the orbits themselves. To give an example of a quotient structure, for the mutual exclusion example shown in Figure 1, observe that swapping the process indices 1 and 2 throughout all states is an automorphism of the structure. Let  $\alpha$  denote this automorphism. Then for this example,  $Aut(\mathcal{M}) = \{\alpha, 1\}$ , where 1 is the identity mapping. Choosing a unique representative from each orbit, we obtain a quotient Kripke structure  $\mathcal{M}_{Aut(\mathcal{M})}$ , as illustrated by Figure 2.

It can be shown [Clarke et al. 1996; Emerson and Sistla 1996] (see Theorem 1, to follow) that a model and its quotient structure satisfy the same *symmetric CTL\** formulas. A *CTL\** formula  $\phi$  is symmetric, or invariant, with respect to  $G$  if for

```

reached := {rep(s) : s ∈ S0};
unexplored := {rep(s) : s ∈ S0};
while unexplored ≠ ∅ do
  remove a state s from unexplored;
  for all successor states q of s do
    if rep(q) is not in reached then
      append rep(q) to reached;
      append rep(q) to unexplored;
    end if
  end for
end while

```

**Fig. 3.** Algorithm to explore a quotient Kripke structure.

every maximal propositional subformula  $f$  appearing in  $\phi$ , and for every  $\alpha \in G$ ,  $\mathcal{M}, s \models f \Leftrightarrow \mathcal{M}, \alpha(s) \models f$ .

**THEOREM 1.** *If  $\mathcal{M}$  and  $\mathcal{M}_G$  denote a model and its quotient model, respectively, with respect to a group  $G$ , then  $\mathcal{M}_G$  is bisimulation equivalent to  $\mathcal{M}$ , and therefore  $\mathcal{M}, s \models \phi \Leftrightarrow \mathcal{M}_G, [s]_G \models \phi$ , for every symmetric  $CTL^*$  formula  $\phi$ .*

As an example, consider the two-process mutual exclusion property “it is not possible for both processes to be in the critical section at the same time.” In terms of the propositions used to label the structures represented by Figures 1 and 2, this property is expressed in  $CTL^*$  as  $A[\neg(C_1 \wedge C_2)]$ . Let us call this property  $\phi_1$ . Clearly,  $\phi_1$  is symmetric with respect to the automorphism group  $\{\alpha, 1\}$ , where  $\alpha$  is defined as earlier. Thus, the Kripke structure represented by Figure 1 satisfies  $\phi_1$  if and only if the quotient structure represented by Figure 2 does. Therefore, to check the mutual exclusion property it is sufficient to check the quotient model only. Note that the quotient model also satisfies the property  $\phi_2$ , defined as  $A[\neg C_2]$ . However, as  $\phi_2$  is not symmetric with respect to the automorphism group, we cannot infer the truth (or otherwise) of  $\phi_2$  for the original Kripke structure (indeed,  $\phi_2$  is clearly not true for the original structure).

The algorithm of Figure 3 (adapted from Ip and Dill [1996]) shows how a quotient structure can be explored incrementally if the symmetries of the Kripke structure can be identified before search. In this case, it may be possible to build the quotient structure even though the original structure is intractably large.

An approach suggested for symmetry reduction during automata-based model checking involves the construction of an *annotated quotient structure* (AQS) [Emerson and Sistla 1996; Sistla 2004]. In this case, there is a labeled edge between representative states  $[s]_G$  and  $[t]_G$  for every edge that exists (in  $\mathcal{M}$ ) from  $rep([s]_G)$  to an element of  $[t]_G$ . If  $(rep([s]_G), t')$  were such an edge (in  $\mathcal{M}$ ), and  $\pi$  the permutation such that  $\pi(rep([t]_G)) = t'$ , then the edge (in the annotated quotient structure) would be labeled with  $\pi$ . Not only is it possible to *unwind* the original structure  $\mathcal{M}$  from the (annotated) quotient structure, but it is also possible to check properties expressed in *indexed  $CTL^*$* —an extension to  $CTL^*$  in which properties include the indexed quantifiers for *all* processes or for *some* process. In addition, the properties to be checked are not required to be symmetric with respect to the group  $G$ . We discuss the use of AQSs to verify properties under fairness assumptions in Section 4.6.2.

### 4.3. Identifying Symmetry

The first step which must be accomplished by any method that exploits symmetry is the identification of symmetries in a model. Let  $\mathcal{M}$  be a Kripke structure. An obvious approach to solving this problem would be to construct  $\mathcal{M}$ , and then to find a symmetry

group  $G$  of  $\mathcal{M}$  using a standard algorithm (e.g., *nauty* [McKay 1981]). These symmetries could be used to reduce  $\mathcal{M}$  to a quotient model,  $\mathcal{M}_G$ .

This approach is flawed in two ways. First, finding the automorphisms of a Kripke structure is equivalent to checking for state-space isomorphism, which for large state-spaces is a hard problem (no polynomial time algorithm is known [McKay 1981]). Second, if enough resources are available to construct  $\mathcal{M}$ , then symmetry reduction is unlikely to be of much benefit. Indeed, the power of reduction techniques is that they allow a reduced model to be checked, even when the unreduced model is intractable.

Thus, the problem is to find symmetries of  $\mathcal{M}$  *without* building  $\mathcal{M}$  explicitly. For simple concurrent programs consisting of a finite number of isomorphic (identical up to renaming) processes executing in parallel, communicating via shared variables, a subgroup of the automorphism group of  $\mathcal{M}$  can be determined from the *communication relation* of the program [Emerson and Sistla 1996]. The communication relation  $CR$  of the program  $\mathcal{P} = \parallel_{i=1}^n p_i$  is defined as the undirected graph  $CR = ([n], E)$ , where  $\{i, j\} \in E$  iff processes  $p_i$  and  $p_j$  share a variable.

**THEOREM 2.** *If  $\mathcal{M}$  is the global state transition state-space of  $\mathcal{P} = \parallel_{i=1}^n p_i$ , where all  $p_i$  are normal and isomorphic, then  $Aut(CR) \leq Aut\mathcal{M}$ .*

The group  $Aut(CR)$  may be automatically computed, since  $CR$  is typically small compared to  $\mathcal{M}$ , or may simply be known in advance.

Theorem 2 applies to systems in which all variables are shared between, at most, two processes, and all processes are of the same type. This result is generalized [Clarke et al. 1998] to remove this restriction via the introduction of the *colored hypergraph*  $HG(\mathcal{P})$  of a shared variable program  $\mathcal{P}$ . The node set of the hypergraph  $HG(\mathcal{P})$  is  $[n]$  and there is a hyperedge  $w \subseteq [n]$  if the program  $\mathcal{P}$  has a variable shared by all process  $p_i$ ,  $i \in w$ . Each node is assigned a color so that two processes  $p_i$  and  $p_j$  are *isomorphic* iff nodes  $i$  and  $j$  have the same color in the colored hypergraph. Two processes are isomorphic in this case if they are of the same process type, and have equivalent sets of transitions.

**THEOREM 3.** *Let  $HG(\mathcal{P})$  be the hypergraph corresponding to the program  $\mathcal{P} = \parallel_{i=1}^n p_i$ . Let  $\mathcal{M}$  be the Kripke structure corresponding to  $\mathcal{P}$ . Given these conditions,  $Aut(HG(\mathcal{P})) \leq Aut(\mathcal{M})$ .*

Another approach to symmetry detection involves the detection of symmetries in the state-space by annotation of the system description via a purpose-built data type [Ip and Dill 1993]. The data type is called a scalarset, which acts as documentation that certain symmetries hold in a specification expressed in the Mur $\phi$  description language [Dill et al. 1992]. A scalarset is an integer subrange with restricted operations as follows:

- An array with a scalarset index type can only be indexed by a scalarset variable of exactly the same type.
- A term of scalarset type must be a variable reference (a scalarset may not appear as an operand to  $+$  or any other operator in a term).
- Scalarset variables may only be compared using  $=$ , and in such cases, must be of exactly the same type.
- For all assignments  $d := t$ , if  $d$  is a scalarset variable,  $t$  must be a term of exactly the same scalarset type.
- If a scalarset variable is used as an index of a **for** statement, the body of the statement is restricted so that the result of the execution is independent of the order of iteration.

The restrictions are sufficient to ensure that the consistent permutation of scalarset variables in all states corresponds to an automorphism of the state-space. Furthermore,

violations of the restrictions can be detected in polynomial time [Ip and Dill 1996]. In the following theorem, a permutation  $\alpha_s$  is a permutation on the elements of scalarset  $s$ , which acts on the states of  $\mathcal{M}$  by permuting state components.

**THEOREM 4.** *Given a source program containing a scalarset  $s$ , every permutation  $\alpha_s$  on the states of the state-space  $\mathcal{M}$  derived from the program is an automorphism of  $\mathcal{M}$ .*

**COROLLARY 1.** *If a program  $\mathcal{P}$  has scalarsets  $s_1, s_2, \dots, s_n$ , there are symmetries in the state graph  $\mathcal{M}$  and we can use the symmetry-reduced state graph  $\mathcal{M}_G$  to perform verification, where  $G$  is the set of all permutations of the states with respect to  $s_1, s_2, \dots, s_n$ .*

An example of the use of scalarsets is in the verification of the Needham-Shroeder public key protocol [Mitchell et al. 1997]. The protocol involves a set of *initiator* processes and a set of *receptor* processes. Each initiator process is identified by the variable *Initiatorid*, which is used to index an array *ini* storing the state of each initiator process. The *Initiatorid* variable is also used as an index within a **for** loop containing the rules determining the behavior of each initiator process. As initiator processes behave symmetrically, by declaring the *Initiatorid* variable a scalarset, symmetry reduction can be automatically performed. Similarly, a scalarset (*Receptorid*) can be used to identify symmetry between receptor processes.

The aforementioned use of scalarsets exploits structural symmetry (symmetry between the processes themselves). The scalarset approach can also be used to exploit *data* symmetry. A scalarset that is used to denote data symmetry is referred to as a *data* scalarset.

**Definition 5.** A scalarset  $s$  is a *data scalarset* in a source program  $\mathcal{P}$  if  $s$  is not used as an array index or a **for** statement index.

If a protocol uses a data scalarset, then it is said to be *data independent* [Wolper 1986]. In this case, symmetry reduction can be used to reduce an infinite state-space (in which data is unbounded) to a finite state-space (with bounded data) thusly:

**THEOREM 5.** *If  $\mathcal{P}$  is a source program, and  $s$  is the name of a data scalarset in  $\mathcal{P}$  and  $\mathcal{P}_1$  and  $\mathcal{P}_2$  are programs identical to  $\mathcal{P}$ , except that  $s$  is declared to be of size  $N_1$  in  $\mathcal{P}_1$  and  $N_2$  in  $\mathcal{P}_2$ , then there exists  $N_s > 0$  such that the symmetry-reduced state graphs of  $\mathcal{P}_1$  and  $\mathcal{P}_2$  are isomorphic whenever  $N_1 \geq N_s$  and  $N_2 \geq N_s$ .*

However, this application of scalarsets is seldom required, as abstraction can be used to eliminate redundant data values [Clarke et al. 1994]. Data symmetry reduction will be discussed again in Section 4.8.

The original scalarset approach [Ip and Dill 1996] only considered the verification of simple safety properties of the form  $AG(\neg error)$ . Nonetheless, scalarsets have been successfully used to exploit symmetry during the verification of more general *LTL* formulas [Bosnacki et al. 2002]. A major drawback to scalarsets is that they only allow the specification of *total* (or *full*) symmetries (where all processes of a given type can be permuted among themselves), thus can be applied to a system of processes connected as a clique, say, but not, for example, as a ring. An alternative data type, called *circularset* [Ip 1996], and additional extensions to scalarset [Donaldson et al. 2005b] have been proposed to handle systems with ring structure and more general systems, respectively. However, these alternatives share with the scalarset approach the problem that the modeler must identify symmetry in the model and use the appropriate data type to specify the presence of this symmetry. This means that symmetry reduction using scalarsets is not a “push-button” reduction technique.

In the message-passing paradigm (in which processes communicate by sending messages on buffered channels), the structural symmetries of a model can be automatically

extracted from program text by using a graph automorphism package such as *nauty* [McKay 1981] or *saucy* [Darga et al. 2004] to analyze the *static channel diagram* (a graphical representation of the communication structure) of the original system [Donaldson et al. 2005a]. This approach is not limited to total symmetries, and has been applied to detect symmetry in PROMELA models [Donaldson and Miller 2005]. A similar approach [Manku et al. 1998] uses GAP [Gap Group 1999] for identifying symmetries in structural descriptions of digital circuits.

#### 4.4. The Orbit Problem

The crux of exploiting symmetry when model checking is that during the search, when a state  $s$  is reached, we must test whether an element  $t$  has already been reached such that  $s = \alpha(t)$  for some  $\alpha \in G$  (i.e.,  $[s]_G = [t]_G$ ). This is known as the *orbit* problem, and central to all model checking methods that exploit symmetry. Techniques must be used either to solve the orbit problem efficiently or to somehow avoid it altogether.

In the design of protocols and sequential circuits, it is common to model a system using a set of Boolean state variables  $x_1, x_2, \dots, x_n$ . In such cases the symmetry group  $G$  is also given in terms of state variables, so that a permutation  $\alpha$  acting on  $\{1, 2, \dots, n\}$  acts on a state vector  $s \in B^n$ , as follows [Clarke et al. 1996]:

$$\alpha((x_1, x_2, \dots, x_n)) = (x_{\alpha(1)}, x_{\alpha(2)}, \dots, x_{\alpha(n)}).$$

*Definition 6.* *The orbit problem* [Clarke et al. 1996] Let  $G$  be a group acting on the set  $\{1, 2, \dots, n\}$ . Given two vectors  $x \in B^n$  and  $y \in B^n$ , the orbit problem is thus: does there exist a permutation  $\alpha \in G$  such that  $y = \alpha(x)$ ?

The orbit problem is related to the graph isomorphism problem:

*Definition 7.* Two graphs  $G_1 = (V_1, E_1)$  and  $G_2 = (V_2, E_2)$  are isomorphic if there is a bijection  $f : V_1 \rightarrow V_2$  such that  $(f(x), f(y)) \in E_2$  if and only if  $(x, y) \in E_1$ . The mapping  $f$  is said to be an isomorphism between  $G_1$  and  $G_2$ .

The issue of determining whether two graphs are isomorphic is known as the *graph isomorphism* problem and is very difficult to solve (although not widely believed to be *NP*-complete). In some cases (e.g., when the maximum degree of the two associated graphs is known to be bounded by a given constant) isomorphism can be determined in polynomial time. However, in general, the graph isomorphism problem is upheld as being highly combinatorially expensive.

**THEOREM 6.** [Clarke et al. 1996] *The orbit problem is as hard as the graph isomorphism problem.*

In fact, the orbit problem has been shown to be equivalent to a well-known issue in computational group theory; the *set stabiliser in a coset* (SSC) problem:

*Definition 8.* Given a set  $X \subseteq [n]$ , a group  $G \subseteq S_n$ , and a group element  $\alpha \in S_n$ , there is an element  $\sigma$  in the coset  $G\alpha = \{\beta\alpha : \beta \in G\}$  which stabilises the set  $X$ , that is,  $\sigma(X) = X$ .

**THEOREM 7.** [Clarke et al. 1998; Jha 1996] *The orbit problem and SSC problem are polynomially equivalent.*

The SSC problem is known [Hoffman 1982; Luks 1991] to be equivalent to several computational group theory problems in *NP*, which are harder than graph isomorphism, but not known to be *NP*-complete.



If BDDs are used to represent the state-space of a model, then exploiting symmetry becomes more complex, as the orbit *relation* of a symmetry group must be represented as a BDD. The orbit relation of a group  $G$  is the set of pairs  $\{(s, t) : t \in [s]\}$ . The BDD of the orbit relation induced by a transitive group (a group acting on a set such that every element of the set is moved by some element of the group) is exponential in the minimum number of components in a system and states in one component [Clarke et al. 1996]. Since transitive groups occur commonly in models of concurrent systems, the combination of standard symmetry reduction techniques with symbolic model checking is limited. We discuss some methods which avoid construction of the orbit relation for symbolic model checking in Section 4.5.

For concurrent programs, modeling using Boolean variables is cumbersome, and it is customary instead to model a state as a vector in  $[k]^n$ , where  $k$  is the size of the set of possible process locations. The action of a symmetry group  $G \subseteq S_n$  on a vector  $(x_1, x_2, \dots, x_n) \in [k]^n$  is analogous to that of a vector in  $B^n$ , as described earlier [Clarke et al. 1996].

When attempting to exploit symmetry, a representative function *rep* is often required, which maps a state  $s$  to the unique representative of its orbit (see Definition 4 and Figure 3). In practice, it is convenient to use the lexicographically least element in the orbit as the representative.

*Definition 9. The Constructive Orbit Problem (COP)* [Clarke et al. 1998; Jha 1996] Given a group acting on  $[n]$  and a vector  $x = (x_1, \dots, x_n) \in [k]^n$ , find the lexicographically least element in the orbit of  $x$ .

**THEOREM 8.** [Clarke et al. 1998; Jha 1996] *The COP is NP-hard.*

In Section 4.5.5 we discuss certain classes of symmetry group for which the COP can be solved in polynomial time.

#### 4.5. Simplifying the Orbit Problem

*4.5.1. Multiple Representatives.* As discussed in the previous section, combining symmetry reduction with symbolic model checking may not be effective, due to the exponentially large BDDs which are required to represent the orbit relation. By using multiple representatives from each orbit, this problem can be avoided to some extent [Clarke et al. 1996, 1998]. If  $G$  is a set of automorphisms, a subset  $C$  of  $G$  is chosen which is closed under inverses and contains the identity element. The set of representatives *Rep* is selected such that each orbit (of the set of states  $S$  under  $G$ ) has at least one element in *Rep*, and for every  $s \in S$ , there is some  $\alpha \in C$  such that  $\alpha(s) \in \text{Rep}$ . The size of *Rep* (and consequently, that of the resulting quotient model) depends heavily on the choice of  $C$ , hence  $C$  must be chosen carefully. The state-space of the quotient model is not reduced (with respect to the original model) as much as with unique representatives. However, multiple representatives reduce the size of the BDDs required to store the state-space, and thus are more effective when the symbolic representation of states is used.

In practice, BDDs reduced through multiple representatives may still be intractably large. Approaches using generic representatives or computing representatives dynamically, which we discuss next in Sections 4.5.2 and 4.5.3, respectively, have been shown to outperform the multiple representatives approach [Emerson and Wahl 2003, 2005a].

*4.5.2. Generic Representatives.* For the symbolic model checking of fully symmetric systems using BDDs, a method which uses *generic* representatives avoids both the orbit problem and construction of the orbit relation [Emerson and Treffer 1999]. This method

involves translating the source program for a model into a reduced program which can be explored using standard model checking algorithms. The idea of generic representatives is best explained using an example. For a basic model of mutual exclusion with three processes (and no token), the states  $(N_1, N_2, T_3)$ ,  $(T_1, N_2, N_3)$ , and  $(N_1, T_2, N_3)$  are all equivalent. This is because there are two processes in the *neutral* local state and one in the *trying* local state in each of the three global states. The generic representative of these states is  $(2N, 1T)$ . A generic representative indicates how many processes are in each local state, but does not refer to individual processes. Hence, the reduced program abstracts from processes to *counters*, with one counter for each local state, which indicates the number of processes currently in that state.

This approach is extended [Emerson and Wahl 2003] to include systems with global shared variables. The translation of a program into reduced form is polynomial in the length of the program and the approach compares well to those using unique or multiple representatives. However, the benefits of this approach can be negated due to the local state explosion problem, where the number of potential local states of a process is exponential in the number of local variables. Since the reduced program requires one counter per local state, BDD representations that require bits to be reserved for each counter become infeasible. Techniques based on *live variable analysis* (similar to the data flow optimizations provided by SPIN [Holzmann 2003]) and *local reachability analysis* [Emerson and Wahl 2005b] have been proposed to limit local state explosion. The generic representatives approach is also very limited, as it only applies to those fully symmetric systems that are simple enough to be amenable to counterabstraction [Emerson and Wahl 2005a].

**4.5.3. Dynamic Computation of Representatives.** Another approach to combining symmetry reduction techniques with symbolic representation (for CTL model checking) involves determining orbit representatives dynamically during fixpoint iterations [Emerson and Wahl 2005a]. Instead of building a representation of the quotient structure for a model, this approach works by computing transition images with respect to the unreduced structure, then mapping the new states to their respective representatives. This approach is not restricted to fully symmetric systems, and can handle data (see Section 4.8) as well as process symmetry. A potential bottleneck here is the operation of swapping bits in the BDD representation of the model, which must be performed repeatedly during representative computation. The complexity of such swaps depends exponentially on the distance, in the BDD variable ordering, between the variables to be swapped. To avoid this problem, permutations are expressed as the product of transpositions of adjacent elements. Experimental results show that this approach outperforms the use of both multiple and generic representatives (see Sections 4.5.1 and 4.5.2, respectively) when applied to a queueing lock algorithm and buggy version of a cache coherence protocol.

**4.5.4. On-the-Fly Representative Selection.** Model checking algorithms that use depth-first search (DFS) can be adapted so that the first element of an orbit encountered during the search is chosen as the orbit representative [Gyuris and Sistla 1999]. However, this approach is not suitable for symbolic model checking techniques, as DFS is very inefficient in the context of BDD state representation. On-the-fly orbit representative selection is possible during breadth-first search (BFS) when the choice of representative is guided using BDD-specific criteria [Barner and Grumberg 2002].

**4.5.5. “Easy” Classes of Groups.** For the following classes of automorphism, group  $G$  (acting on a model of a system of  $n$  processes), the constructive orbit problem (COP)

can be solved in polynomial time [Clarke et al. 1998; Jha 1996]:

- $G$  has order polynomial in  $n$ , for example, a cyclic or dihedral group, or the group associated with an  $n \times n$  torus
- $G$  is the full symmetric group  $S_n$
- $G$  is a disjoint or wreath product of groups for which the COP is polynomial time solvable
- $G$  is generated by transpositions

Note that when  $G$  is the full symmetric group  $S_n$ , the lexicographically least (lex-least) element of the orbit of a state can be obtained by sorting the state-vector. When  $G$  has order polynomial in  $n$ , the COP can be solved by enumerating the orbit of a state. In the other cases, the lex-least element is found by sorting the segments of the state-vector individually.

*4.5.6. Using Orbit Representatives in Practice.* The scalarset method [Ip and Dill 1996] assumes the existence of a canonicalization function (in which states are replaced by a unique equivalence class representative) or normalization function (in which states of a subset are used as multiple representative states). For symmetry reduction in Mur $\phi$  a suitable canonicalization function [Ip and Dill 1993] applies all permutations to a state  $s$  and returns the lexicographically smallest image. An approach using a normalization function is also suggested, in which the state-vector is split into two parts. For a given state, a permutation  $\phi$  is selected that produces the lexicographically smallest image of the first (most significant) part of the associated state-vector. The representative state chosen is the concatenation of the image of the two parts of the state-vector (under  $\phi$ ).

The use of normalization and canonicalization functions with scalarsets is extended [Bosnacki et al. 2001] using heuristics to choose the order in which variables are positioned in the state-vector. This ordering determines, for example, which variables are most significant and appear in the first (leftmost) part of the split state-vector. One approach, the *sorted* strategy, involves the identification of an array that is indexed by a scalarset type (the *main* array), and placing it in the leftmost position of the state-vector. In another approach, the *segmented* strategy, the lexicographically smallest image of the second part of the state-vector, with respect to all permutations that canonicalize the first part, is used in the representative state. There is tradeoff between reduction in memory requirements and faster verification for the sorted and segmented strategies. The segmented strategy yields canonical representatives, but is more computationally expensive than the sorted strategy. On the other hand, use of the sorted strategy may result in exploring several states from the same equivalence class.

Two further approaches, *pc-sorted* and *pc-segmented*, are also suggested for systems in which no suitable main arrays exist, but the process identities are of type scalarset. In this case, a main array is constructed that contains the program counters. A prototype implementation of this approach is implemented in the SymmSpin package [Bosnacki et al. 2000, 2002], which we discuss in Section 5.1.

A canonicalization function is suggested, again within the context SPIN [Nalumasu and Gopalakrishnan 1995], for systems with any (user-defined) symmetry. Though less restrictive than the scalar-set approach (full symmetry is not required and more general operations on permutable variables are permitted), a unique canonicalization function must be constructed manually by the modeler for every individual model, thereby limiting the applicability of the method.

#### 4.6. Combining Symmetry Reduction with Other Techniques

Basic symmetry reduction does not take into account the more sophisticated techniques associated with model checking. In this section we discuss how symmetry reduction can be safely combined with partial order reduction, and modified to successfully handle fairness.

*4.6.1. Symmetry and Partial Order Reduction.* Partial order (see Section 2.3.6) and symmetry reduction are orthogonal reduction techniques. They can therefore be successfully used in conjunction, resulting in larger savings in memory and verification time.

The combination of these two techniques was first suggested in the context of Petri nets [Valmari 1989]. This approach applies to the stubborn sets method of partial order reduction and is restricted to deadlock detection.

The idea of combining two reductions simultaneously is extended to verifying next-time free *LTL* properties via model checking [Emerson et al. 1997]. Indeed, an algorithm is given combining partial order reduction and any bisimulation-preserving equivalence. When the equivalence is the automorphism relation, the algorithm proceeds as follows: From any state  $s$ , an ample set of transitions is calculated. The orbit representatives of any states that are reachable via these transitions are then explored. A similar algorithm, combining the persistent sets method of partial order reduction with symmetry reduction, is used within the stateless search technique implemented in VeriSoft [Godefroid 1999].

*4.6.2. Exploiting Symmetry Under Fairness Assumptions.* Fairness is vital for proving liveness properties, as it reflects the basic requirement that processes are executing at an indefinite yet positive speed [Emerson and Sistla 1997]. Two important kinds are *weak* and *strong* fairness. Given a Kripke structure  $\mathcal{M}$ , an infinite path  $\pi$  of  $\mathcal{M}$  is *strongly fair* if each process that is enabled infinitely often is executed infinitely often. A path  $\pi$  is *weakly fair* if any process that is continuously enabled is executed infinitely often.

Fairness is generally incompatible with basic symmetry reduction methods because the progress of an individual process along a path of the quotient structure cannot be tracked in the usual way. Each state of the quotient structure is labeled according to the representative of an equivalence class of states in the original structure, and for a transition  $s \rightarrow t$  in the original model, it may not be the case that transition  $rep([s]_G) \rightarrow rep([t]_G)$  also occurs in the quotient model.

These fundamental problems are overcome when the automata-theoretic approach using annotated quotient structures (see Section 4.2) is used in the context of (inherently symmetric) fair-indexed *CTL\** properties [Emerson and Sistla 1997; Sistla 2004]. An annotated quotient structure  $\mathcal{M}_G$  is used together with an automaton  $\mathcal{A}$  that accepts only fair computations. An efficient algorithm, based on finding maximal strongly connected components (MSCCs) [Tarjan 1972] (see Section 2.3.2), is presented for model checking fair-indexed *CTL\** formulas under the assumption of strong and (by implication) weak fairness. Correctness results (including liveness properties) are verified for a resource controller example using a prototype (fair) model checker. Comparison with an unreduced model indicates an exponential reduction in the number of stored states.

This approach to symmetry-reduced model checking has been extended to the on-the-fly case [Gyuris and Sistla 1999] in which  $\mathcal{M}_G \times \mathcal{A}$  is checked during construction. The approach also exploits *state symmetries* [Emerson and Sistla 1996]. A state symmetry of a state  $s$  is a permutation  $\alpha \in Aut(\mathcal{M})$  on process indices such that  $\alpha(s) = s$ . If processes  $i$  and  $j$  have the same local state in global state  $s$ , and if  $\alpha(i) = j$ , then only the transitions made from state  $s$  by process  $i$  need to be considered, thus saving space and computation time. The resulting algorithm is exploited in the symmetry-based model checker (SMC) [Sistla et al. 2000], which we discuss in Section 5.1.

A parallel approach to model checking with symmetry reduction and weak fairness [Bosnacki 2003] combines the weak fairness algorithm implemented in SPIN [Holzmann 2003] (based on the Choeka flag algorithm [Choueka 1974]) with a symmetry reduction algorithm [Bosnacki 2002] based on the nested depth-first search (NDFS) approach to model checking [Holzmann et al. 1996]. As well as exploiting the usual advantages over MSCC algorithms, the NDFS approach is compatible with approximate verification techniques, such as the hash-compact method and bitstate hashing (see Section 2.4.1).

#### 4.7. Exploiting Symmetry in Less Symmetric Systems

Many systems which occur commonly in practice are comprised of several *similar*, but not all identical, processes. An example is the readers-writers problem [Emerson and Trefler 1999], where  $m$  reader processes and  $n$  writer processes access a shared resource, for some  $m, n > 0$ .

A writer always has priority over a reader when both are trying to access the shared resource. If  $\mathcal{M}$  is a model of this system, then  $\mathcal{M}$  is not fully symmetric. In fact,  $\text{Aut}(\mathcal{M}) = S_{\{1,2,\dots,m\}} \times S_{\{m+1,m+2,\dots,m+n\}}$ , that is, both readers and writers can be permuted, but readers cannot be interchanged with writers.<sup>1</sup> However, the state graph is symmetric in every sense, except for transitions from a state where two processes are attempting to access the shared resource.

It is possible to exploit this kind of *almost* symmetry during model checking. Indeed, by defining different classes of “symmetry,” such as *near* or *rough* symmetry [Emerson and Trefler 1999] and *virtual* symmetry [Emerson et al. 2000], it is still possible to infer temporal logic properties of the system by model checking a suitable quotient graph using the entire group  $S_{m+n}$  as the automorphism group.

Suppose  $\mathcal{M}$  is a model of a system, and  $\mathcal{I}$  the set of process identifiers associated with  $\mathcal{M}$ . Then, a permutation  $\alpha \in \text{Sym } \mathcal{I}$  is said to be a near automorphism of  $\mathcal{M}$  if for every transition  $s \rightarrow t$  of  $\mathcal{M}$ , either  $\alpha(s) \rightarrow \alpha(t)$  is a transition of  $\mathcal{M}$  or  $s$  is totally symmetric with respect to  $\text{Aut}(\mathcal{M})$ , that is,  $s$  is invariant under  $\text{Aut}(\mathcal{M})$ . The model  $\mathcal{M}$  is said to be nearly symmetric if it has a suitable group of near automorphisms  $G_n$ .

If, on the other hand,  $G_r$  is a subgroup of  $\text{Sym } \mathcal{I}$ , then  $\mathcal{M}$  is roughly symmetric with respect to  $G_r$  if for every pair of states  $s$  and  $s'$ , where  $s \sim_{G_r} s'$ , any transition from  $s$  is matched by a transition from  $s'$ , provided the associated local transition (from  $s'$ ) involves a process with the highest priority. If  $\mathcal{M}$  is a nearly (roughly) symmetric model with respect to group  $G_n$  ( $G_r$ ), then despite the lack of complete symmetry, the quotient model  $\mathcal{M}_{G_n}$  ( $\mathcal{M}_{G_r}$ ) is bisimilar to the original model  $\mathcal{M}$ . It follows that symmetry reduction preserves all symmetric  $CTL^*$  properties.

Both near and rough symmetry are subsumed by the notions of *virtual* and *strong* virtual symmetry [Emerson et al. 2000]. In addition to systems with static priorities (which can already be described via rough symmetry), virtual symmetry applies to systems where the resources are asymmetrically shared according to dynamic priorities.

The symmetrization  $R^G$  of a transition relation  $R$  by a group  $G$  is defined by

$$R^G = \{\alpha(s) \rightarrow \alpha(t) : \alpha \in G \text{ and } s \rightarrow t \in R\}.$$

Intuitively, symmetrizing a transition relation can be considered to be the process of adding transitions that are missing due to asymmetry in the system.

A structure  $\mathcal{M}$  is said to be virtually symmetric with respect to a group  $G_v$  acting on  $S$  if for any  $s \rightarrow t \in R^{G_v}$ , there exists  $\alpha \in G_v$  such that  $s \rightarrow \alpha(t) \in R$ . In addition, if for any  $s \rightarrow t \in R^{G_v}$  there exists  $\alpha$  in  $\text{Fix}(s, G_v)$  (the largest subgroup of  $G_v$  which

<sup>1</sup>Assuming there are no symmetries other than those which permute process ids.

fixes  $s$ ) such that  $s \rightarrow \alpha(t) \in R$ , then  $\mathcal{M}$  is said to be strongly virtually symmetric with respect to  $G_v$ . If a Kripke structure  $\mathcal{M}$  is (strongly) virtually symmetric with respect to a group  $G_v$ , then  $\mathcal{M}$  is bisimilar to the quotient model  $\mathcal{M}_{G_v}$ , and model checking of symmetric properties can be performed over  $\mathcal{M}_{G_v}$ . A procedure is given to identify the case where a Kripke structure is strongly virtually symmetric with respect to a group  $G_v$ . This procedure involves the local counting of transitions that are present in  $R^{G_v}$ , but absent in  $R$ . Virtual symmetry has been successfully combined with the generic representatives approach (see Section 4.5.2) for the case where processes are fully interchangeable with respect to virtual symmetry [Wei et al. 2005]. This allows symmetry-reduced symbolic model checking of partially symmetric systems by using the NuSMV model checker [Cimatti et al. 2002] (see Section 2.4.2).

A method involving the symmetry reduction of models that have little or no symmetry uses guarded annotated quotient structures (GQSs) [Sistla 2004; Sistla and Godefroid 2004]. These structures are an extension to the annotated quotient structures [Emerson and Sistla 1996, 1997; Gyuris and Sistla 1999] discussed in Section 4.2. Suppose  $\mathcal{M}$  is the Kripke structure of a system, and  $\mathcal{M}' \supseteq \mathcal{M}$  is obtained from  $\mathcal{M}$  by adding transitions (in a similar manner to the process of symmetrization described previously), so that  $\mathcal{M}'$  has more symmetry than  $\mathcal{M}$ . A guarded annotated quotient structure for  $\mathcal{M}$  can be viewed as an annotated quotient structure for  $\mathcal{M}'$ , with edges labeled to indicate which processes can make the transition (in  $\mathcal{M}$ ). Thus, the original edges of  $\mathcal{M}$  can be recovered from the representation of  $\mathcal{M}'$ . A temporal formula  $\phi$  can be checked over the guarded annotated quotient structure by unwinding the structure, even if  $\phi$  is not symmetric with respect to the automorphisms used for reduction. This approach potentially allows large factors of reduction to be obtained, since a larger group of automorphisms is used than would be possible using standard quotient structure reduction. Indeed, experimental results using the SMC model checker [Sistla et al. 2000] show how the GQS method can be applied effectively to a system of prioritized processes.

A recent extension to the GQS approach [Sistla et al. 2004] involves (symmetry-reduced) model checking of *extended CTL* (*CCTL*) properties (which involve an additional construct, *COUNT*, for specifying the number of components in a given state). This extended logic is more expressive than indexed *CTL* (see Section 4.2).

Properties are again not restricted to being fully symmetric in an alternative automata-theoretic approach [Ajami et al. 1998], but must be *partially* symmetric. For example, consider the following property: “If some process is waiting for a resource then it will get it, provided none of the processes with higher identity will require the resource in the future.” To check the satisfaction of a formula  $\phi$  for a model  $\mathcal{M}$  with set of states  $S$ , a set of equivalence relations is first computed between states of  $\mathcal{B}$ , the Büchi automaton representing  $\phi$ . If  $G$  is a symmetry group of  $\mathcal{M}$ , one equivalence relation is defined for every element of  $G$ . Two states,  $b_1, b_2 \in \mathcal{B}$ , are equivalent with respect to  $\alpha \in G$  if and only if the predecessors and successors of  $b_1$  are mapped to the predecessors and successors of  $b_2$ , respectively (and vice versa). The quotient graph is then constructed by applying the equivalence relations to the pairs of states  $(s, b) \in S \times \mathcal{B}$ . The approach is extended [Haddad et al. 2000] to partially symmetric *models* by representing the model itself as the synchronized product of a symmetric model and an asymmetric Büchi automaton. The method is illustrated using well-formed Petri nets.

#### 4.8. Exploiting Data Symmetry

Most of the symmetry reduction methods described in this article relate to structural symmetry. However, as discussed in Section 4.3, another form of symmetry, namely data symmetry, can be exploited to increase the effectiveness of model checking. In Section 4.3 we discussed the application of scalarsets to exploit data symmetries.

As software specifications often involve large data structures with vast numbers of values, it may be impossible to check whether properties hold for every feasible assignment of values to the dataset. That is, it may not be possible to check the properties for every interpretation of the model. It is therefore desirable only to check representative models for each equivalence class of interpretations.

This use of data equivalence is exploited for software analysis using the Nitpick specification tool [Jackson et al. 1998].

## 5. IMPLEMENTATIONS OF SYMMETRY REDUCTION

In this section we list the major tools for which symmetry reduction has been implemented. This is not intended as an exhaustive exposition, but as selective illustration.

### 5.1. Explicit State Methods

*Mur $\phi$* . The *Mur $\phi$*  description language is the first to have been augmented with the *scalarset* data type (see Section 4.3). As a result, the *Mur $\phi$*  verification system [Dill et al. 1992] (see Section 2.4) is the first to implement symmetry reduction using scalarsets [Ip and Dill 1996] and has inspired many of the other implementations discussed in this section.

An automorphism group for the state-space is determined statically from the *Mur $\phi$*  description and consists of all permutations that permute *scalarset* variables. The lexicographically smallest member of each orbit is used as the orbit representative and a suitable canonicalization function (see Section 4.5.6) is used to map every state to its orbit representative.

*Mur $\phi$*  has been used to verify a number of highly symmetric algorithms (e.g., Peterson's  $n$ -process mutual exclusion algorithm [Peterson 1981]) and a lock implementation for the Stanford DASH multiprocessor [Lenoski et al. 1992].

The *Mur $\phi$*  tool has been extended with two alternative classes of algorithm for representative computation [Juntilla 2004]. The first class of algorithms transforms each state encountered during the search to a *characteristic* graph, and derives a canonical state representative from the canonical form of this graph. The *nauty* graph isomorphism tool [McKay 1981] is used to perform canonicalization operations. The other class of algorithms uses ordered partitions on states, and during canonicalization considers only permutations which are compatible with the partitioning of a given state. This approach mimics the partitioning approach commonly used by graph isomorphism algorithms [McKay 1981].

*SMC*. The symmetry-based model checker (SMC) [Sistla 2004; Sistla et al. 2000] is an explicit state model checker which has been specifically designed for the verification of highly symmetric systems. Exploiting both *process* and *state* symmetry, in addition to proving safety properties, SMC is the only model checker that can be used to effectively verify liveness properties under both strong and weak fairness assumptions. Model checking is performed using a technique [Gyuris and Sistla 1999] involving annotated quotient structures (AQSs) (see Sections 4.2 and 4.6.2). The AQS can be constructed either in advance or on-the-fly. For on-the-fly construction, it is also possible to store the edges of the AQS during construction. If the edges are not stored, considerable space savings can be made. However, verification time is increased dramatically.

The input language of SMC uses a syntax similar to that of *Mur $\phi$*  [Dill et al. 1992]. Processes are separated into modules such that all processes in a given module are identical, up to renaming (note that these modules are analogous to the *scalarsets* used by *Mur $\phi$* ). Symmetry cannot be exploited in programs where there is not total symmetry within each component type, for example, in a token ring network.

The AQS is constructed incrementally, and the first state of an orbit encountered during search is used as the representative for that orbit. State symmetries of a state  $s$  are detected by partitioning processes within each module into equivalence classes. A *leader* process is chosen from each equivalence class, and only transitions from  $s$  made by one of the leader processes are explored. Reached states are stored in a hash table, and a hashing function is used which always hashes equivalent states to the same location, and desirably hashes inequivalent states to different locations. For a state  $s$ , the hashing function returns  $Checksum(s) \bmod b$ , where  $b$  is the size of the hash table. The checksum for a state is computed from the values of variables in that state. Each time a state is to be stored at a position in the hash table, a check is made to see if the state is equivalent to any other state in that position in the table. Two states with differing checksums cannot be equivalent, so SMC performs the pretest of comparing checksums before checking the equivalence of two states. In many cases, this quickly shows the nonequivalence of states.

To check whether two states with equal checksums are equivalent, a polynomial time bounded, randomized algorithm is used which runs in quadratic time. This algorithm sometimes falsely reports that two equivalent states are not equivalent, which may result in the construction of a larger-than-optimal AQS (but this is not unsafe).

SMC has been used to check the correctness of the link-layer part of the IEEE standard 1394 “Firewire” protocol [IEEE-1394 1995], and also a resource controller example. The latter shows that exploiting state symmetry can speed up verification considerably when the number of processes is high. Recent extensions of SMC [Sistla and Godefroid 2004; Sistla et al. 2004] enable partially symmetric systems with priorities to be verified over a GQS, and the properties to be expressed in an extended form of *CTL*.

*SymmSpin*. Symmetric SPIN (SymmSpin) [Bosnacki et al. 2002] is a symmetry reduction package for the SPIN model checker [Holzmann 2003] (see Section 2.4.1). To allow the process symmetry of a system to be specified, the scalarset data type [Ip and Dill 1996] is used. However, to avoid modifying the PROMELA parser, rather than directly extending PROMELA language with the scalarset data type, all symmetry information is provided (by the user) in a separate file. This is called the *system description* file and identifies which variables have the scalarset type.

For a given Promela model, SPIN generates a verifier for it in the form of a C program which is compiled and executed. SymmSpin modifies this program to add symmetry reduction via a *representative* function, which for a given state computes an orbit representative for that state. For a given orbit, the representative is the least element with respect to a specified canonicalization function or one of the minimal elements computed via a normalization function (see Section 4.5.6). During search, SymmSpin stores the original states on the stack and representative states on the heap (see Section 2.3.2). This means that the counterexample traces generated by SymmSpin correspond to real counterexample traces through the model, rather than representatives of a counterexample trace.

Experiments using SymmSpin show that for certain models, the factor of reduction gained is close to the theoretical limit [Bosnacki et al. 2002]. They also show that the combination of symmetry and partial order reduction can be effective. A prototype extension of SymmSpin for symmetry-reduced model checking under weak fairness [Bosnacki 2003] has recently been developed. This is discussed in Section 4.6.2.

*Other SPIN-Based Implementations*. An extension to SPIN is proposed [Derepas and Gastin 2001] to allow symmetry reduction of models of systems of replicated processes. The specification language PROMELA is augmented with two additional keywords, *ref* and *public*, which identify reference variables and local variables with public scope, respectively. Unlike scalarsets, these variables may hold the addresses of other processes for communication purposes or may represent process ids. Orbit representatives



are computed by a process called *pseudosorting* in which the parts of the state-vector corresponding to individual processes are sorted lexicographically. As the original state-vector ordering depends on the order in which variables are declared, the efficiency of the sorting algorithm depends on the initial declaration ordering. Only fully symmetric properties can be verified using this technique.

An on-the-fly state-space exploration algorithm exploiting both process and heap object symmetry has been implemented in the dSPIN model checking tool [Iosif 2002] (see Section 2.4.4). For dynamic systems modeled using dSPIN, the number of state components may grow along an execution path. Therefore, rather than applying symmetry reduction with respect to a fixed permutation group, a family of groups is considered. A suitable group is selected at each execution step. Orbit representatives are calculated using a similar set of heuristics to that used by SymmSpin.

The SymmExtractor tool [Donaldson and Miller 2005] can be used to detect structural symmetries arising from the communication structure of a PROMELA model (see Section 4.3).

## 5.2. Symbolic Methods

*SMV*. As a symbolic model checker, SMV [McMillan 1993] does not lend itself to symmetry reduction of the state-space. This is because the symbolic representation of the orbit relation as a BDD is prohibitively large (see Section 4.4). However, symmetry reduction on the *cases* associated with a property to be proved for a system is achieved via the use of scalarsets [McMillan 2000]. In order to exploit the abstraction techniques available with SMV, a method called *temporal case-splitting* is used to break a given property down into a parameterized set of assertions. This addresses state explosion, but may result in an unwanted side-effect, namely, *case explosion*. Declaring variables as scalarsets enables SMV to sort assertions into equivalence classes. Specifically, if we have two assertions  $\phi_1$  and  $\phi_2$ , where  $\phi_2$  is obtained from  $\phi_1$  by some permutation of scalarset values, then  $\phi_1$  holds if and only if  $\phi_2$  holds. Thus for a given parameterized set of assertions, it is only necessary to check a representative subset of assertions. This representative subset is chosen in such a way that every assertion in the original parameterized set can be mapped to a representative assertion via permutation of scalarset values.

*SYMM*. One (purpose-built) symbolic model checker that exploits symmetry reduction methods for the verification of *CTL* specifications is SYMM [Clarke et al. 1998]. SYMM uses a simple input language that is based on a shared variable model of computation, and allows the user to give symmetries of the system to be verified.

To combat the orbit problem, symmetry reduction is implemented using the *multiple orbit representatives* approach (see Section 4.5.1). SYMM has been used to verify the IEEE Futurebus arbiter protocol [IEEE-896.1 1992], which controls a number of prioritized components competing for a resource. Each individual process is described via a *module*. Modules with the same priority can be permuted.

*Other Symbolic Implementations*. The RuleBase model checker [Beer et al. 1996] (see Section 2.4.2) has been experimentally extended with symmetry reduction techniques for *under-approximation* [Barner and Grumberg 2002]. Generators for a symmetry group of the verified system are supplied by the user. The generators which are genuine symmetries of the system, and under which the checked property is invariant, are retained by the model checker for exploitation during search. Orbit representatives are selected on-the-fly (see Section 4.5.4). Experimental results show that RuleBase performs significantly better for checking liveness properties when symmetry reduction is applied. However, no improvement in performance has been shown for safety properties.

An experimental model checking system, UTOOL [Emerson and Wahl 2005b], has been developed for the investigation of techniques to combine symmetry reduction with symbolic representation. This tool uses the input language of Mur $\phi$  and performs symbolic verification, exploiting symmetry wherever possible. UTOOL avoids constructing the orbit relation through the use of generic representatives or dynamic representative computation (see Sections 4.5.2 and 4.5.3, respectively). Though less efficient, for the purposes of comparison, UTOOL also implements symmetry reduction using precomputed multiple representatives (see Section 4.5.1).

### 5.3. Real-Time Methods

*UPPAAL*. The real-time model checking tool, UPPAAL, has been extended to exploit symmetry [Hendriks et al. 2003] by using scalarsets [Ip and Dill 1996]. As the main purpose of UPPAAL is to perform reachability analysis, symmetry reduction using scalarsets is an obvious choice—the original scalarset theory was developed in the context of reachability analysis, rather than, for checking temporal logic properties. However, the soundness of symmetry reduction does not follow directly, since the UPPAAL language is very different from that of Mur $\phi$ . Hence, soundness is proved separately for UPPAAL.

The implementation of symmetry reduction in UPPAAL involves the development of an efficient algorithm for the computation of a canonical representative for a state. This is particularly challenging because UPPAAL represents sets of clock valuations symbolically using a difference-bounded matrix (DBM).

The scalarsets for a given model define a set of state *swaps* for the model. Each state swap is an automorphism of the model, and the set of all state swaps can be used to compute a canonical state representative. In order to simplify the computation of representatives, two assumptions are made. The first is that an array indexed by scalarsets does not contain elements of scalarset type. The second is that a timed automaton in a UPPAAL model may only reset its clock to the value zero. This assumption ensures that individual clocks can always be ordered using the order in which they were reset—this is called the *diagonal* property and leads to a total ordering on states. Note that the diagonal property is important, since for a given total ordering, minimization using the state swaps of a general DBM is at least as hard as testing isomorphism for strongly regular graphs [Hendriks et al. 2003].

A state is minimized using the state swaps defined by scalarsets in the model, together with this total ordering. This minimized state is a canonical representative for the original state.

Experimental results for Fischer’s mutual exclusion protocol, presented in some detail, show that exponential savings can be gained by exploiting symmetry. Further experiments for an audio/video protocol, and for a distributed agreement algorithm, are also encouraging. Since symmetry reduction in UPPAAL makes use of scalarsets, only total symmetries can be exploited.

*RED*. Another (symbolic) real-time model checker to support symmetry reduction is RED [Wang and Schmidt 2002]. The symmetry reduction algorithm uses relations between pointers to define an ordering among processes. This ordering is then used to compute a representative by sorting the associated orbits. Every permutation is constructed via the successive composition of transpositions. This can lead to an over-approximation of the reachable state-space (the “anomaly of image false reachability”). For this reason, using RED with symmetry reduction is only useful for checking that a state is *not* reachable. The performance of RED (with symmetry reduction) is compared to that of Mur $\phi$  [Dill et al. 1992] (with symmetry reduction) and SMC [Sistla et al. 2000] for three benchmark systems [Wang and Schmidt 2002]. Since it manages to

successfully combine symbolic techniques with symmetry reduction, as the number of processes increases, RED dramatically outperforms the other model checkers.

#### 5.4. Direct Model Checking

*Bogor.* A symmetry reduction technique has been developed for the Bogor model checking framework [Robby et al. 2003], which is used to model check Java programs (see Section 2.4.4). The symmetry reduction methods used in Bogor [Robby et al. 2003] are based on those implemented in dSPIN [Iosif 2002] (see Section 5.1), but use more efficient heuristics [Iosif 2004] for state-vector sorting.

States contain both thread and heap information. These different parts of the state (the thread and the heap state) are sorted separately. Threads are sorted by comparing associated program counters. This does not always produce a unique ordering. However, heap states can be sorted in a canonical way. For every heap state  $s$ , there is an associated set of memory locations,  $l_{1,s}, l_{2,s}, \dots, l_{r,s}$ , say. It is possible to sort the indices of the memory locations (for a given  $s$ ) by ordering the traces associated with each pair  $(s, l_{i,s})$ ,  $1 \leq i \leq r$ . The trace for pair  $(s, l_{i,s})$  is the smallest of all the incoming *chains* (pairs of thread identifiers and variable sequences) that can themselves be ordered in a natural way. The sorting of the location indices produces a strictly ordered list of integers. If  $G$  is a symmetry group acting on the heap elements, then the ordered list associated with state  $s$  is identical to the corresponding list for any  $s'$  in the same orbit of  $G$  as  $s$ . Thus, the index sorting function is a canonicalization function (see Section 4.5.6).

*VeriSoft.* The VeriSoft model checker [Godefroid 1997] verifies  $C$ -code directly via a stateless search (see Section 2.4.4). As such, the symmetry reduction methods implemented in VeriSoft [Godefroid 1999] rely on equivalences between sequences of *transitions*, rather than between states. If  $\mathcal{M}$  is a transition system and  $\mathcal{M}_G$  a quotient transition system of  $\mathcal{M}$  with respect to the equivalence of transition sequences, then  $\mathcal{M}$  and  $\mathcal{M}_G$  are bisimilar and satisfy the same (symmetric) temporal logic formulas.

In order for equivalent transitions to be identified, labels are added to transitions so that the model is represented by a labeled transition system. Two transitions are equivalent with respect to a given symmetry group  $G$  if their respective labels are equivalent with respect to  $G$ . This concept can be easily extended to sequences of transitions. Symmetry reduction is used to prune transitions on-the-fly. If, for some  $\alpha \in G$ , transitions  $t$  and  $\alpha(t)$  are enabled and  $\alpha$  fixes  $s$ , then only one of  $t$  or  $\alpha(t)$  need be explored. Given that  $s$  is not stored explicitly, it is not straightforward to check that  $\alpha$  fixes  $s$ . However, assuming that  $\alpha$  fixes the initial state  $s_0$ , if  $w$  is the sequence of transitions leading from  $s_0$  to  $s$ , then it can be shown that  $\alpha(s) = s$  if and only if  $w$  and  $\alpha(w)$  are equivalent with respect to a partial ordering of transitions. Thus, by combining symmetry reduction with partial order reduction techniques (see Section 4.6.1), the problem of checking that  $\alpha(s) = s$  is overcome.

*Other Direct Model Checking Implementations.* A limited form of symmetry reduction is applied [Lerda and Visser 2001] within the second, generation Java PathFinder tool (JPF2) [Visser et al. 2000] (see Section 2.4.4), which model checks Java bytecode directly. Like dSPIN, JPF2 is capable of handling dynamic structures (although unlike dSPIN, data is not allocated dynamically). States are composed of a static area, dynamic area, and thread area, each of which is represented as an array. Two states are considered to be equivalent if a permutation applied to the static and dynamic area arrays of the first state gives the corresponding arrays of the second. A canonicalization function (see Section 4.5.6) is used, which imposes a simple ordering (calculated during model checking) on the static and dynamic areas of the states.

## 6. CONCLUSION

Model checking algorithms rely upon the construction of a model representing all system states. One of the major problems associated with model checking is state-space explosion. The main approaches to overcoming state-space explosion involve a reduction in either state representation size (e.g., symbolic representations) or the number of states or paths explored (e.g., on-the-fly methods, partial order reduction and symmetry reduction). Symmetry reduction involves avoiding areas of the state-space which are symmetrically equivalent to those already visited. In this article we have given an overview of symmetry reduction, and how it relates to other reduction approaches.

The identification of symmetries involves finding symmetries of a model without building the model explicitly. Purpose-built data types, such as scalarsets and circularsets, allow permutations to be identified that correspond to automorphisms. However, they can only be used for systems where subsets of processes behave identically, whereas many computer science applications naturally involve *partial* symmetries, in which individual processes are distinguished in some way. Recent results extend to partial symmetries, as well as to the automatic extraction of symmetry from source programs, in certain cases.

The crux of exploiting symmetry is the orbit problem—it must be either solved efficiently or avoided altogether. This makes symmetry reduction ineffective, in general, for symbolic model checking. As a result, its implementation within the most widely used symbolic model checker, SMV, is very limited. However, the use of multiple representatives, generic representatives, or dynamic representative computation makes the combination of symmetry reduction and symbolic techniques theoretically possible. Symmetry is amenable to combination with partial order reduction and also to fairness when using automata-theoretic approaches.

Most implementations of symmetry reduction using scalarsets are within mainstream on-the-fly checkers such as Mur $\phi$  and SPIN, though scalarsets have also been added to the real-time model checking tool UPPAAL. Other techniques have been developed to implement symmetry within the specialist symbolic model checker RED and within model checkers that are used to verify C or Java code directly (e.g., VeriSoft, Bogor, and Java PathFinder (JPF2)).

Open problems remain, including the development of improved techniques to deal with partial symmetries [Emerson 2000], identification of (full or partial) symmetries from Java and C source programs, and the identification and exploitation of symmetry in probabilistic model checking.

## REFERENCES

- AJAMI, K., HADDAD, S., AND ILIE, J. 1998. Exploiting symmetry in linear time temporal logic model checking: One step beyond. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '98)*, B. Steffen, Ed. Lisbon, Portugal. Lecture Notes in Computer Science, vol. 1384. Springer-Verlag, 52–67.
- ALUR, R., COURCOUBETIS, G., AND DILL, D. 1990. Model-Checking for real-time systems. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*. Philadelphia, PA. IEEE Computer Society Press, 414–425.
- ALUR, R. AND DILL, D. 1993. A theory of timed automata. *Inf. Comput.* 194, 2–34.
- ALUR, R. AND HENZINGER, T. 1992. Logics and models of real time: A survey. In *Proceedings of the REX Workshop on Real-Time: Theory and Practice*. Mook, the Netherlands. Lecture Notes in Computer Science, vol. 600. Springer-Verlag, 74–106.
- ALUR, R. AND KURSHAN, R. 1995. Timing analysis in COSPAN. In *Proceedings of the 3rd DIMACS/SYCON Workshop on Hybrid Systems: Verification and Control*, R. Alur et al., Eds. Lecture Notes in Computer Science, vol. 1066. New Brunswick, NJ. Springer-Verlag, 220–231.

- BALL, T., COOK, B., LEVIN, V., AND RAJAMANI, K. 2004. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In *Proceedings of the 4th International Conference on Integrated Formal Methods (IFM 2004)*, E. Boiten et al. Eds. Canterbury, UK. Lecture Notes in Computer Science, vol. 2999. Springer-Verlag, 1–20.
- BALL, T. AND RAJAMANI, S., EDs. 2003. *Model Checking Software: Proceedings of the 10th International SPIN Workshop (SPIN 2003)*. Portland, OR. Lecture Notes in Computer Science, vol. 2648. Springer-Verlag.
- BARNER, S. AND GRUMBERG, O. 2002. Combining symmetry reduction and under-approximation for symbolic model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV 2002)*. Copenhagen, Denmark. Lecture Notes in Computer Science, vol. 2404. Springer-Verlag, 93–106.
- BARRETT, G. 1995. Model checking in practice: The t9000 virtual channel processor. *IEEE Trans. Softw. Eng.* 21, 2, 69–78.
- BEER, I., BEN-DAVID, S., EISNER, C., AND LANDVER, A. 1996. Rulebase: An industry-oriented formal verification tool. In *Proceedings of the 33rd Conference on Design Automation (DAC '96)*. Las Vegas, NV. ACM Press, 655–660.
- BEN-DAVID, S. AND HEYMAN, T. 2000. Scalable distributed on-the-fly symbolic model checking. In *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD 2000)*, W. A. Hunt Jr. and S. D. Johnson, Eds. Austin, TX. Lecture Notes in Computer Science, vol. 1954. Springer-Verlag, 390–404.
- BEST, B. AND GRAHLMANN, B. 1996. PEP—More than a Petri net tool. In *Proceedings of the 2nd International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '96)*, T. Margaria and B. Steffen, Eds. Passau, Germany. Lecture Notes in Computer Science, vol. 1055. Springer-Verlag, 397–401.
- BEST, B. AND KOUTNY, M. 1995. A refined view of the box calculus. In *Proceedings of the 16th International Conference on the Application and Theory of Petri Nets (ATPN '95)*, G. De Michelis and M. Diaz, Eds. Turin, Italy. Lecture Notes in Computer Science, vol. 935. Springer-Verlag, 103–118.
- BHAT, G., CLEAVELAND, R., AND GRUMBERG, O. 1995. Efficient on-the-fly model checking for  $CTL^*$ . In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science*. San Diego, CA. IEEE Computer Society Press, 388–397.
- BOLLIG, B. AND WEGENER, I. 1996. Improving the variable ordering of OBDDs is NP-complete. *IEEE Trans. Comput.* 45, 9, 993–1002.
- BOLOGNESI, T. AND BRINKSMA, E. 1987. Introduction to the ISO specification language LOTOS. *Comput. Netw. ISDN Syst.* 14, 1, 25–59.
- BOSNACKI, D. 2002. A nested depth-first search algorithm for model checking with symmetry reduction. In *Proceedings of the 22nd International Conference on Formal Techniques for Networked and Distributed System (FORTE 2002)*. Houston, TX. Lecture Notes in Computer Science, vol. 2529. Springer-Verlag, 40–56.
- BOSNACKI, D. 2003. A light-weight algorithm for model checking with symmetry reduction and weak fairness. In *Model Checking Software: Proceedings of the 10th International SPIN Workshop (SPIN 2003)*. T. Ball and S. Rajamani, Eds. Portland, OR. Lecture Notes in Computer Science, vol. 2648. Springer-Verlag, 89–103.
- BOSNACKI, D., DAMS, D., AND HOLENDERSKI, L. 2000. Symmetric Spin. In *Proceedings of the 7th SPIN Workshop (SPIN 2000)*, K. Havelund et al., Eds. Stanford, CA. Lecture Notes in Computer Science, vol. 1885. Springer-Verlag, 1–19.
- BOSNACKI, D., DAMS, D., AND HOLENDERSKI, L. 2001. A heuristic for symmetry reductions with scalarsets. In *Proceedings of the International Symposium of Formal Methods Europe (FME 2001)*, J. N. Oliveira and Z. Pamela, Eds. Berlin, Germany. Lecture Notes in Computer Science, vol. 2021. Springer-Verlag, 518–533.
- BOSNACKI, D., DAMS, D., AND HOLENDERSKI, L. 2002. Symmetric Spin. *Int. J. Soft. Tools Technol. Transfer* 4, 1, 65–80.
- BRINKSMA, E. AND LARSEN, K., EDs. 2002. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV 2002)*. Copenhagen, Denmark. Lecture Notes in Computer Science, vol. 2404. Springer-Verlag.
- BRYANT, R. 1992. Symbolic Boolean manipulation with ordered binary decision diagrams. *ACM Comput. Surv.* 24, 3, 293–318.
- BURCH, J., CLARKE, E., McMILLAN, K., DILL, D., AND HWANG, L. 1992. Symbolic model checking:  $10^{20}$  states and beyond. *Inf. Comput.* 98, 2, 142–170.
- CALDER, M. AND MILLER, A. 2001. Using SPIN for feature interaction analysis—A case study. In *Proceedings of the 8th International SPIN Workshop (SPIN 2001)*, M. Dwyer, Ed. Toronto, Canada. Lecture Notes in Computer Science, vol. 2057. Springer-Verlag, 143–162.

- CALDER, M. AND MILLER, A. 2003. Generalizing feature interactions in email. In *Feature Interactions in Telecommunications and Software Systems VII*, D. Amyot and L. Logrippo, Eds. IOS Press, Ottawa, Canada, 187–205.
- CATTEL, T. 1994. Modeling and verification of a multiprocessor real-time OS kernel. In *Proceedings of the 7th WG6.1 International Conference on Formal Description Techniques (FORTE '94)*, D. Hogrefe and S. Leue, Eds. Berne, Switzerland. International Federation for Information Processing, vol. 6. Chapman and Hall, London, UK, 55–70.
- CHOUKEA, Y. 1974. Theories of automata on  $\omega$ -tapes: A simplified approach. *J. Comput. Sys. Sci.* 8, 117–141.
- CIMATTI, A., CLARKE, E., GIUNCHIGLIA, E., GIUNCHIGLIA, F., PISTORE, M., ROVERI, M., SEBASTIANI, R., AND TACHELLA, A. 2002. NuSMV2: A new OpenSource tool for symbolic model checking. In *Proceedings of the 14th International Conference on Computer-Aided Verification (CAV 2002)*, E. Brinksma and K. Larsen, Eds. Copenhagen, Denmark. Lecture Notes in Computer Science, vol. 2404. Springer-Verlag, 359–364.
- CIMATTI, A., CLARKE, E., GIUNCHIGLIA, F., AND ROVERI, M. 1999. NuSMV: A new symbolic model verifier. In *Proceedings of the 11th International Conference on Computer-Aided Verification (CAV '99)*, N. Halbwegs and D. Peled, Eds. Trento, Italy. Lecture Notes in Computer Science, vol. 1633. Springer-Verlag, 495–499.
- CIMATTI, A., GIUNCHIGLIA, F., MINGARDI, G., ROMANO, D., TORIELLI, F., AND TRAVERSO, P. 1997. Model checking safety critical software with SPIN: An application to a railway interlocking system. In *Proceedings of the 3rd SPIN Workshop (SPIN '97)*, R. Langerak, Ed. Twente University, The Netherlands, 5–17.
- CLARKE, E., EMERSON, E., JHA, S., AND SISTLA, A. 1998. Symmetry reductions in model-checking. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV '98)*, A. Hu and M. Vardi, Eds. Vancouver, BC, 147–158.
- CLARKE, E., EMERSON, E., AND SISTLA, A. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.* 8, 2, 244–263.
- CLARKE, E., ENDERS, R., FILKHORN, T., AND JHA, S. 1996. Exploiting symmetry in temporal logic model checking. *Formal Methods Syst. Desi.* 9, 1–2, 77–104.
- CLARKE, E., GRUMBERG, O., AND LONG, D. 1994. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.* 16, 5, 1512–1542.
- CLARKE, E., GRUMBERG, O., AND PELED, D. 1999. *Model Checking*. MIT Press, Cambridge, MA.
- CLARKE, E. AND WING, J. M. 1996. Formal methods: State-of-the-art and future directions. *ACM Comput. Surv.* 28, 4, 626–643. Report by the Working Group on Formal Methods for the ACM Workshop on Strategic Directions in Computing Research.
- CLAYBERG, E. AND RUBEL, D. 2004. *Eclipse: Building Commercial-Quality Plug-Ins*. Addison Wesley, Reading, MA.
- CORBETT, J., DWYER, M., HATCLIFF, J., LAUBACH, S., PĂSRĂEANU, C., ROBBY, AND ZHENG, H. 2000. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*. Limerick, Ireland. ACM Press, New York, 439–448.
- COURCOUBETIS, C., VARDI, M., WOLPER, P., AND YANNAKAKIS, M. 1992. Memory-Efficient algorithms for the verification of temporal properties. *Formal Methods in Syst. Des.* 1, 275–288. Also appeared in *Proceedings of the 2nd International Workshop on Computer-Aided Verification (CAV '90)*, 207–218.
- DAMS, D., GERTH, R., LEUE, S., AND MASSINK, M. EDs. 1999. In *Proceedings of the 5th and 6th International SPIN Workshops*. Trento, Italy and Toulouse, France. Lecture Notes in Computer Science, vol. 1680. Springer-Verlag.
- DANJANI-BROWN, S., COFER, D., HARTMANN, G., AND PRATT, S. 2003. Formal modeling and analysis of an avionics triplex sensor voter. In *Model Checking Software: Proceedings of the 10th International SPIN Workshop (SPIN 2003)*, T. Ball and S. Rajamani, eds, Portland, OR. Lecture Notes in Computer Science, vol. 2648. Springer-Verlag. 34–48.
- DARGA, P., LIFFITON, M., SAKALLAH, K., AND MARKOV, I. 2004. Exploiting structure in symmetry detection for CNF. In *Proceedings of the 41st Annual Conference on Design Automation*. San Diego. CA. ACM Press, New York, 530–534.
- DE BAKKER, J. W., HUIZING, C., DE ROEVER, W., AND ROZENBERG, G., EDs. 1992. In *Proceedings of the REX Workshop on Real-Time: Theory and Practice*. Mook, the Netherlands. Lecture Notes in Computer Science, vol. 600. Springer-Verlag.
- DEMARTINI, C., IOSIF, R., AND SISTO, R. 1999. A deadlock detection tool for concurrent Java programs. *Softw. Pract. Exper.* 29, 7, 577–603.
- DEREPAS, F. AND GASTIN, P. 2001. Model checking systems of replicated processes with Spin. In *Proceedings of the 8th International SPIN Workshop (SPIN 2001)*, M. Dwyer, Ed. Toronto, Canada. Lecture Notes in Computer Science, vol. 2057. Springer-Verlag, 235–251.

- Dijkstra, E. 1976. *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, NJ.
- DILL, D. 1996. The Mur $\phi$  verification system. In *Proceedings of the 8th International Conference on Computer Aided Verification (CAV '96)*, R. Alur and T. Henzinger, Eds. New Brunswick, NJ. Lecture Notes in Computer Science, vol. 1102. Springer-Verlag, 390–393.
- DILL, D., DREXLER, A., HU, A., AND YANG, C. H. 1992. Protocol verification as a hardware design aid. In *Proceedings of the 1992 IEEE International Conference on Computer Design: VLSI in Computer and Processors (ICCD'92)*, Cambridge, MA, IEEE Computer Society, 522–525.
- DONALDSON, A. AND MILLER, A. 2005. Automatic symmetry detection for model checking using computational group theory. In *Proceedings of the 13th International Symposium on Formal Methods (FM 2005)*, J. Fitzgerald et al., Eds. Newcastle, UK. Lecture Notes in Computer Science, vol. 3582. Springer-Verlag, 481–496.
- DONALDSON, A., MILLER, A., AND CALDER, M. 2005a. Finding symmetry in models of concurrent systems by static channel diagram analysis. *Electron. Notes Theoretical Comput. Sci.* 128, 6, 161–177.
- DONALDSON, A., MILLER, A., AND CALDER, M. 2005b. SPIN-to-GRAPE: A tool for analysing symmetry in Promela models. *Electron. Notes Theoretical Comput. Sci.* 139, 1, 3–23.
- DWYER, M., ED. 2001. In *Proceedings of the 8th International SPIN Workshop (SPIN 2001)*. Toronto, Canada. Lecture Notes in Computer Science, vol. 2057. Springer-Verlag.
- EMERSON, E. 1992. Real time and the  $\mu$ -calculus. In *Proceedings of the REX Workshop on Real-Time: Theory and Practice*, I. W. de Bakker et al., Eds. Mook, the Netherlands. Lecture Notes in Computer Science, vol. 600. Springer-Verlag, 176–194.
- EMERSON, E. 2000. Model checking: Theory into practice. In *Proceedings of the 20th International Conference on Foundations of Software Technology and Theoretical Computer Science*, S. Kapoor and S. Prasad, Eds. New Delhi, India. Lecture Notes in Computer Science, vol. 1974. Springer-Verlag, 1–10.
- EMERSON, E., HAVLICEK, J., AND TREFLER, R. 2000. Virtual symmetry reduction. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science*. Santa Barbara, CA. IEEE Computer Society Press, 121–131.
- EMERSON, E., JHA, S., AND PELED, D. 1997. Combining partial order and symmetry reductions. In *Proceedings of the 3rd International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '97)*, E. Brinksma, Ed. Enschede, the Netherlands. Lecture Notes in Computer Science, vol. 1217. Springer-Verlag, 19–34.
- EMERSON, E. AND LEI, C. 1987. Modalities for model checking: Branching time logic strikes back. *Sci. Comput. Program.* 8, 3, 275–306.
- EMERSON, E. AND SISTLA, A. 1996. Symmetry and model checking. *Formal Methods Syst. Des.* 9, 1–2, 105–131.
- EMERSON, E. AND SISTLA, A. 1997. Utilizing symmetry when model-checking under fairness assumptions: An automata-theoretic approach. *ACM Trans. Program. Lang. Syst.* 19, 4, 617–638.
- EMERSON, E. AND TREFLER, R. 1999. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME '99)*, L. Pierre and T. Kropf, Eds. Bad Herrenalp, Germany. Lecture Notes in Computer Science, vol. 1703. Springer-Verlag, 142–156.
- EMERSON, E. AND WAHL, T. 2003. On combining symmetry reduction and symbolic representation for efficient model checking. In *Proceedings of the 12th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2003)*, D. Geist and E. Tronci, Eds. L'Aquila, Italy. Lecture Notes in Computer Science, vol. 2860. Springer-Verlag, 216–230.
- EMERSON, E. AND WAHL, T. 2005a. Dynamic symmetry reduction. In *Proceedings of the 11th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2005)*. Held as part of the Joint European Conference on Theory and Practice of Software (ETAPS 2005), N. Halbwachs and L. Zuck, Eds. Edinburgh, UK. Lecture Notes in Computer Science, vol. 3440. Springer-Verlag, 382–396.
- EMERSON, E. AND WAHL, T. 2005b. Efficient reduction techniques for systems with many components. *Electron. Notes Theoretical Comput. Sci.* 130, 379–399.
- GAP GROUP. 1999. *GAP—Groups Algorithms and Programming, Version 4.2*. Aachen, St. Andrews. <http://www-gap.dcs.st-and.ac.uk/~gap>.
- GARAVEL, H. AND SIFAKIS, J. 1990. Compilation and verification of LOTOS specifications. In *Proceedings of the IFIP WG6.1 10th International Symposium on Protocol Specification, Testing and Verification (PSTV '90)*, L. Logrippo et al., Eds. Ottawa, Canada. 379–394.
- GIRAULT, C. AND VALK, R., EDs. 2003. *Petri Nets for Systems Engineering: A Guide to Modeling, Verification, and Applications*. Springer-Verlag, New York.

- GODEFROID, P. 1996a. On the costs and benefits of using partial-order methods for the verification of concurrent systems. In *Proceedings of the DIMACS Workshop on Partial-Order Methods, in Verification (POMIV '96)*, D. Peled et al., Eds. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 29. American Mathematical Society, Boston, MA. 289–303.
- GODEFROID, P. 1996b. *Partial Order Methods for the Verification of Concurrent Systems*. Lecture Notes in Computer Science, vol. 1032. Springer-Verlag, Berlin.
- GODEFROID, P. 1997. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th Symposium on Principles of Programming Languages (POPL '97)*. Paris, France. ACM Press, New York, 174–186.
- GODEFROID, P. 1999. Exploiting symmetry when model-checking software (extended abstract). In *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification (FORTE/PSTV '99)*, J. Wu et al., Eds. Beijing, China. International Federation for Information Processing, vol. 156. Kluwer, 257–275.
- GREGOIRE, J.-C., HOLZMANN, G., AND PELED, D., EDs. 1996. *Proceedings of the 2nd Workshop on the SPIN Verification System*. Rutgers University, NJ. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 32. American Mathematical Society, Boston, MA.
- GYURIS, V. AND SISTLA, A. 1999. On-the-fly model checking under fairness that exploits symmetry. *Formal Methods Syst. Des.* 15, 3, 217–238.
- HADDAD, S., ILIE, J., AND AJAMI, K. 2000. A model checking method for partially symmetric systems. In *Proceedings of the 20th IFIP International Conference on Formal Description Techniques/Protocol Specification, Testing, and Verification (FORTE/PSTV 2000)*. Pisa, Italy. International Federation For Information Processing. Kluwer, 121–136.
- HAVELUND, K. AND PRESSBURGER, T. 2000. Model checking JAVA programs using JAVA Pathfinder. *Inte. J. Softw. Tools Technol. Transfer* 2, 4, 366–381.
- HENDRIKS, M., BEHRMANN, G., LARSEN, K., NIEBERT, P., AND VAANDRAGER, F. 2003. Adding symmetry reduction to UPPAAL. In *Proceedings of the 1st International Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS 2003)*, K. Larson and P. Niebert, Eds. Merseille, France. Lecture Notes in Computer Science, vol. 2791. Springer-Verlag, 46–59.
- HENZINGER, T., HO, P., AND WONG-TOI, H. 1997. HyTech: A model checker for hybrid systems. *Int. J. Softw. Tools Technol. Transfer* 1, 1/2 (December), 110–122.
- HENZINGER, T., JHALA, R., MAJUMDAR, R., AND SUTRE, G. 2003. Software verification with BLAST. In *Model Checking Software: Proceedings of the 10th International SPIN Workshop (SPIN 2003)*, T. Ball and S. Rajamani, Eds. Portland, OR. Lecture Notes in Computer Science, vol. 2648. Springer-Verlag. 235–239.
- HILLSTON, J. 1996. *A Compositional Approach to Performance Modeling*. Distinguished Dissertations in Computer Science. Cambridge University Press, Cambridge, UK.
- HOFFMAN, C. 1982. *Group Theoretic Algorithms and Graph Isomorphism*. Lecture Notes in Computer Science, vol. 136. Springer-Verlag, Berlin.
- HOGREFE, D. AND LEUE, S., EDs. 1994. In *Proceedings of the 7th WG6.1 International Conference on Formal Description Techniques (FORTE '94)*. Berne, Switzerland. International Federation for Information Processing, vol. 6. Chapman and Hall, London, UK.
- HOLZMANN, G. 1998. An analysis of bitstate hashing. *Formal Methods Syst. Des.* 13, 3, 289–307.
- HOLZMANN, G. 1999. The engineering of a model checker: The Gnu i-protocol case study revisited. In *Proceedings of the 5th and 6th International SPIN Workshops*. D. Dams et al., Eds. Trento, Italy and Toulouse, France. Lecture Notes in Computer Science, vol. 1680. Springer-Verlag. 232–244.
- HOLZMANN, G. 2003. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, Boston, MA.
- HOLZMANN, G. AND JOSHI, R. 2004. Model-Driven software verification. In *Proceedings of the 11th International SPIN Workshop (SPIN 2004)*, S. Graf and L. Mounier, Eds. Barcelona, Spain. Lecture Notes in Computer Science, vol. 2989. Springer-Verlag. 76–91.
- HOLZMANN, G. AND PELED, D. 1994. An improvement in formal verification. In *Proceedings of the 7th WG6.1 International Conference on Formal Description Techniques (FORTE '94)*, D. Hogrefe and S. Leue, Eds. Berne, Switzerland. International Federation for Information Processing, vol. 6. Chapman and Hall, London, UK. 197–211.
- HOLZMANN, G., PELED, D., AND YANNAKAKIS, M. 1996. On nested depth first search. In *Proceedings of the 2nd Workshop on the SPIN Verification System*, J.-C. Gregoire et al., Eds. Rutgers University, NJ. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 32. American Mathematical Society, Boston, MA. 23–32.



- HOLZMANN, G. AND SMITH, M. 1999a. A practical method for the verification of event-driven software. In *Proceedings of the 21st International Conference on Software engineering (ICSE '99)*. Los Angeles, CA. ACM Press, New York, 597–607.
- HOLZMANN, G. AND SMITH, M. 1999b. Software model checking—Extracting verification models from source code. In *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification (FORTE/PSTV '99)*, J. Wu et al., Eds. Beijing, China. International Federation for Information Processing, vol. 156. Kluwer, 481–497.
- HU, A. AND VARDI, M., EDs. 1998. *Proceedings of the 10th International Conference on Computer-Aided Verification (CAV '98)*. Vancouver, BC. Lecture Notes in Computer Science, vol. 1427. Springer-Verlag.
- HUBER, P., JENSON, A., JEPSON, L., AND JENSON, K. 1984. Towards reachability trees for high-level Petri nets. In *Proceedings of the European Workshop on Applications and Theory in Petri Nets*. G. Rozenberg et al., Eds. Aarhus, Denmark. Lecture Notes in Computer Science, vol. 188. Springer-Verlag, 215–233.
- IEEE-1394. 1995. *IEEE Standard for a High Performance Serial Bus Std 1394–1995*. Institute of Electrical and Electronic Engineers.
- IEEE-896.1. 1992. *IEEE Standard for Futurebus+—Logical Protocol Specification Std 896.1-1991*. Institute of Electrical and Electronic Engineers.
- IOSIF, R. 2002. Symmetry reduction criteria for software model checking. In *Proceedings of the 9th International SPIN Workshop (SPIN 2002)*, D. Bosnacki and S. Leue, Eds. Grenoble, France. Lecture Notes in Computer Science, vol. 2318. Springer-Verlag, 22–41.
- IOSIF, R. 2004. Symmetry reductions for model checking of concurrent dynamic software. *Int. J. Softw. Tools Technol. Transfer* 6, 4, 302–319.
- IOSIF, R. AND SISTO, R. 1999. dSPIN: A dynamic extension of SPIN. In *Proceedings of the 5th and 6th International SPIN Workshops*. D. Dams et al., Eds. Trento, Italy and Toulouse, France. Lecture Notes in Computer Science, vol. 1680. Springer-Verlag. 20–33.
- IP, C. AND DILL, D. 1996. Better verification through symmetry. *Formal Methods in Syst. Des.* 9, 41–75.
- IP, C. N. 1996. State reduction methods for automatic formal verification. Ph.D. thesis, Department of Computer Science, Stanford University.
- IP, C. N. AND DILL, D. L. 1993. Better verification through symmetry. *Comput. Hardw. Description Lang. Appl.* A-32, 97–111.
- JACKSON, D., JHA, S., AND DAMON, C. 1998. Isomorph-Free model enumeration: A new method for checking relational specifications. *ACM Trans. Program. Lang. Syst.* 20, 2, 302–343.
- JHA, S. 1996. Symmetry and induction in model checking. Ph.D. thesis, School of Computer Science, Carnegie Mellon University.
- JUNTILLA, T. 2004. New orbit algorithms for data symmetries. In *Proceedings of the 4th International Conference on Application of Concurrency to System Design (ACSD 2004)*. Ontario, Canada. IEEE Computer Society, 175–184.
- KUMAR, S. AND LI, K. 2002. Using model checking to debug device firmware. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI 2002)*. USENIX, Boston, MA.
- KURSHAN, R. 1995. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton Series in Computer Science. Princeton University Press, Princeton, NJ.
- KWIATKOWSKA, M., NORMAN, G., AND PARKER, D. 2002. Probabilistic symbolic model checking with PRISM. In *Proceedings of the 8th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2002)*. Held as part of the Joint European Conference on Theory and Practice of Software (ETAPS 2002), J. Katoen and P. Stevens, Eds. Grenoble, France. Lecture Notes in Computer Science, vol. 2280. Springer-Verlag, 52–66.
- LARSON, K., PETERSSON, P., AND YI, W. 1995. Model-checking for real-time systems. In *Proceedings of the 10th International Symposium on the Fundamentals of Computation Theory (FCT '95)*, H. Reichel Ed. Dresden, Germany. Lecture Notes in Computer Science, vol. 965. Springer-Verlag, 62–88.
- LARSON, K., PETERSSON, P., AND YI, W. 1997. UPPAAL in a nutshell. *Int. J. Softw. Tools. Technol. Transfer* 1, 1/2, 134–152.
- LENOSKI, D., LAUDON, J., GHARACHORLOO, K., WEBER, W.-D., GUPTA, A., HENNESSEY, J., HOROWITZ, M., AND LAM, M. 1992. The directory-based cache coherence protocol for the DASH multiprocessor. *IEEE Comput.* 25, 3, 63–79.
- LÉONARD, L. AND LEDUC, G. 1997. An introduction to ET-LOTOS for the description of time-sensitive systems. *Comput. Netw. ISDN Syst.* 29, 3, 271–292.
- LÉONARD, L. AND LEDUC, G. 1998. A formal definition of time in LOTOS. *Formal Aspects Comput.* 10, 3, 248–266.

- LERDA, F. AND VISSER, W. 2001. Addressing dynamic issues of program model checking. In *Proceedings of the 8th International SPIN Workshop (SPIN 2001)*, M. Dwyer, Ed. Toronto, Canada. Lecture Notes in Computer Science, vol. 2057. Springer-Verlag, 80–102.
- LICHTENSTEIN, O. AND PNUELLI, A. 1985. Checking that finite state concurrent programs satisfy their linear specification. In *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages (POPL '85)*. New Orleans, LA. ACM Press, New York, 97–107.
- LUKS, E. 1991. Permutation groups and polynomial-time computation. In *Groups and Computation*, L. Finkelstein and W. Kantor, Eds. DIMACS series in Discrete Mathematics and Theoretical Computer Science, vol. 11. American Mathematical Society, Boston, MA, 139–176.
- MANKU, G., HOJATI, R., AND BRAYTON, R. 1998. Structural symmetry and model checking. In *Proceedings of the 10th International Conference on Computer-Aided Verification (CAV '2002)*, A. Hu and M. Vardi, Eds. Vancouver, BC. Lecture Notes in Computer Science, vol. 1427. Springer-Verlag, 159–171.
- MATESCU, R. 2003. On-the-Fly verification using CADP. In *Proceedings of the 8th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2003)*, T. Arts and W. Fokink, Eds. Trondheim, Norway. Electronic Notes in Theoretical Computer Science, vol. 80. Elsevier, 1–5.
- MATESCU, R. AND GARAVEL, H. 1998. XTL: A metalanguage and tool for temporal logic model-checking. In *Proceedings of the International Workshop on Software Tools for Technology Transfer (STTT '98)*, T. Margaria and B. Steffen, Eds. Aalborg, Denmark.
- McKAY, B. D. 1981. Practical graph isomorphism. *Congressus Numerantium* 30, 45–87.
- McMILLAN, K. 1993. *Symbolic Model Checking*. Kluwer Academic, Hingham, MA.
- McMILLAN, K. 2000. A methodology for hardware verification using compositional model checking. *Sci. Comput. Program.* 37, 1–3, 279–309.
- McMILLAN, K. L. AND SCHWALBE, J. 1992. Formal specification of the Gigamax cache consistency protocols. In *Proceedings of the 1991 International Symposium on Shared Memory Multiprocessors*, N. Suzuki, Ed. Tokyo. Information Processing Society of Japan. MIT Press, 242–251.
- MERZ, S. 2000. Model checking: A tutorial overview. In *Modeling and Verification of Parallel Processes, 4th Summer School (MOVEP 2000)*, F. Cassez et al., Eds. Nantes, France. Lecture Notes in Computer Science, vol. 2067. Springer-Verlag, 3–38.
- MITCHELL, J., MITCHELL, M., AND STERN, U. 1997. Automated analysis of cryptographic protocols using Mur $\phi$ . In *Proceedings of the IEEE Symposium on Security and Privacy*. Oakland, CA. IEEE Computer Society, 141–151.
- MÜLLER-OLM, M., SCHMIDT, D., AND STEFFEN, B. 1999. Model-Checking: A tutorial introduction. In *Proceedings of the 6th International Static Analysis Symposium (SAS '99)*, A. Cortesi and G. File, Eds. Venice, Italy. Lecture Notes in Computer Science, vol. 1694. Springer-Verlag, 330–354.
- NALUMASU, R. AND GOPALAKRISHNAN, G. 1995. Explicit-Enumeration-Based verification made memory-efficient. In *Proceedings of the 12th IFIP International Conference on Computer Hardware Description Languages and Their Applications (CHDL '95)*, S. D. Johnston, Ed. Chiba, Japan. Elsevier Science, 617–622.
- NICOLLIN, X. AND SIFAKIS, J. 1994. ATP: Theory and application. *Inf. Comput.* 114, 131–178.
- PELED, D. 1996a. Combining partial order reductions with on-the-fly model checking. *Formal Methods Syst. Des.* 8, 39–64.
- PELED, D. 1996b. Partial order reduction: Linear and branching temporal logics and process algebras. In *Proceedings of the DIMACS Workshop on Partial-Order Methods in Verification (POMIV '96)*, D. Peled et al., Eds. Princeton, NJ. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 29. American Mathematical Society, Boston, MA. 233–257.
- PELED, D., PRATT, V., AND HOLZMANN, G., EDS. 1996. *Proceedings of the DIMACS Workshop on Partial-Order Methods in Verification (POMIV '96)*. Princeton, NJ. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 29. American Mathematical Society, Boston, MA.
- PELED, D. AND VARDI, M., EDS. 2002. *Proceedings of the 22nd International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2002)*. Houston, TX. Lecture Notes in Computer Science, vol. 2529. Springer-Verlag.
- PETERSON, G. 1981. Myths about the mutual exclusion problem. *Inf. Process. Lett.* 12, 3, 115–116.
- PNUELLI, A. 1981. The temporal semantics of concurrent programs. *Theoretical Comput. Sci.* 13, 45–60.
- QUIELLE, J. AND SIFAKIS, J. 1982. Specification and verification of concurrent systems in CÆSAR. In *Proceedings of the 5th International Symposium on Programming*, M. Dezani-Ciancaglini and U. Montanari, Eds. Torino, Italy. Lecture Notes in Computer Science, vol. 137. Springer-Verlag, 195–220.
- ROBBY, DWYER, M., AND HATCLIFF, J. 2003. Bogor: An extensible and highly-modular model checking framework. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*.

- Held jointly with *the 9th European Software Engineering Conference (ESEC/FSE 2003)*. Helsinki, Finland. ACM Press, New York, 267–276.
- ROBBY, DWYER, M., HATCLIFF, J., AND IOSEF, R. 2003. Space-Reduction strategies for model checking dynamic software. *Electron. Notes Theoretical Comput. Sci.* 89, 3, 499–517.
- RUTTEN, J., KWAITOWSKA, M., NORMAN, G., AND PARKER, D. 2004. *Mathematical Techniques for Analysing Concurrent and Probabilistic Systems*. CRM Monograph Series, vol. 23. American Mathematical Society, Centre de Recherches Mathématiques, Université de Montréal.
- SCHNEIDER, K. 2003. *Verification of Reactive Systems*. Springer-Verlag, Berlin.
- SISTLA, A. 2004. Employing symmetry reductions in model checking. *Comput. Lang. Syst. Structures* 3, 99–137.
- SISTLA, A. AND GODEFROID, P. 2004. Symmetry and reduced symmetry in model checking. *ACM Trans. Program. Lang. Syst.* 25, 4, 702–734.
- SISTLA, A., GYURIS, V., AND EMERSON, E. 2000. SMC: A symmetry-based model checker for verification of safety and liveness properties. *ACM Trans. Softw. Eng. Methodol.* 9, 133–166.
- SISTLA, A., WANG, X., AND ZHOU, M. 2004. Checking extended CTL properties using guarded quotient structures. In *Proceedings of the 2nd IEEE International Conference on Software Engineering and Formal Methods*. IEEE Computer Society, 87–94.
- STARKE, P. H. 1991. Reachability analysis of Petri nets using symmetries. *Syst. Anal. Model. Simul.* 8, 4/5, 293–303.
- TARJAN, R. 1972. Depth-first search and linear graph algorithms. *SIAM J. Comput.* 1, 2, 146–160.
- VALMARI, A. 1989. Stubborn sets for reduced state space generation. In *Proceedings of the 10th International Conference on Application and Theory of Petri Nets*. Bonn, Denmark. Lecture Notes in Computer Science, vol. 483. Springer-Verlag, 491–515.
- VALMARI, A. 1992. A stubborn attack on state explosion. *Formal Methods Syst. Des.* 1, 297–322.
- VARDI, M. AND WOLPER, P. 1986. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings of the 1st Annual IEEE Symposium on Logic in Computer Science*. Cambridge, MA. IEEE Computer Society Press, 332–344.
- VARDI, M. AND WOLPER, P. 1994. Reasoning about infinite computations. *Inf. Comput.* 115, 1–37.
- VARPAANIEMI, K., HALME, J., HIEKKANEN, K., AND PYSSYSALO, T. 1995. PROD reference manual. Tech. Rep. B13, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland.
- VERGAUWEN, B. AND LEWI, J. 1993. A linear local model checking algorithm for CTL. In *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR '93)*, E. Best, Ed. Hildesheim, Germany. Lecture Notes in Computer Science, vol. 715. Springer-Verlag, 447–461.
- VISSER, W. AND BARRINGER, H. 1996. Memory efficient state storage in SPIN. In *Proceedings of the 2nd Workshop on the SPIN Verification System*, J.-C. Gregoire et al., Eds. Rutgers University, NJ. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 32. American Mathematical Society, Boston, MA. 185–203.
- VISSER, W., HAVELUND, K., BRAT, G., AND PARK, S. 2000. Model checking programs. In *Proceedings of the 15th IEEE Conference on Automated Software Engineering (ASE-2000)*, P. Alexander and P. Flener, Eds. Grenoble, France. IEEE Computer Society Press, 3–12.
- WANG, F. AND SCHMIDT, K. 2002. Symmetric symbolic safety-analysis of concurrent software with pointer data structures. In *Proceedings of the 22nd International Conference on Formal Techniques for Networked and Distributed System (FORTE 2002)*, D. Peled and M. Vardi, Eds. Houston, TX. Lecture Notes in Computer Science, vol. 2529. Springer-Verlag, 50–64.
- WEI, O., GURFINKEL, A., AND CHECHIK, M. 2005. Identification and counter abstraction for full virtual symmetry. In *Proceedings of the 13th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME '05)*, D. Borrione and W. J. Paul, Eds. Saarbrücken, Germany. Lecture Notes in Computer Science, vol. 3725. Springer-Verlag, 285–300.
- WOLPER, P. 1986. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages (POPL '86)*. St. Petersburg Beach, FL. ACM Press, New York, 184–193.
- WOLPER, P. AND LEROY, D. 1993. Reliable hashing without collision detection. In *Proceedings of the 5th International Conference on Computer Aided Verification (CAV '93)*, C. Courcoubetis, Ed. Elounda, Greece. Lecture Notes in Computer Science, vol. 697. Springer-Verlag, 59–70.
- WOLPER, P., VARDI, M., AND SISTLA, A. 1983. Reasoning about infinite computation paths. In *Proceedings of the 4th IEEE Symposium on Foundations of Computer Science*. Tucson, AZ. IEEE Computer Society, 185–194.

- WU, J., CHANSON, S., AND GAO, Q., EDS. 1999. *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification (FORTE/PSTV '99)*. Beijing, China. International Federation for Information Processing, vol. 156. Kluwer.
- YI, W., PETTERSSON, P., AND DANIELS, M. 1994. Automatic verification of real-time communicating systems by constraint-solving. In *Proceedings of the 7th WG6.1 International Conference on Formal Description Techniques (FORTE '94)*, D. Hogrefe and S. Leue, Eds. Berne, Switzerland. International Federation for Information Processing, vol. 6. Chapman and Hall, London, UK, 243–258.
- YOVINE, S. 1997. Kronos: A verification tool for real-time systems. *Int. J. Softw. Tools Technol. Transfer* 1, 1/2, 123–133.
- YUEN, C. AND TJIOE. 2001. Modeling and verifying a price model for congestion control in computer networks using Promela/SPIN. In *Proceedings of the 8th International SPIN Workshop (SPIN 2001)*, M. Dwyer, Ed. Toronto, Canada. Lecture Notes in Computer Science, vol. 2057. Springer-Verlag, 272–287.

Received December 2004; revised November 2005; accepted March 2006