SRI International

CSL Technical Note • June 1, 2005

Discovering Symmetries

Hassen Saïdi
Computer Science Laboratory
SRI International
Menlo Park, CA 94025

This research was partially sponsored by DARPA under contract number N66001-00-C-8058. The views herein are those of the authors and do not necessarily reflect the views of the supporting agency. This research was also partially sponsored by NSF under contract number SA4102-10097PG/CCR-0325274

Computer Science Laboratory • 333 Ravenswood Ave. • Menlo Park, CA 94025 • (650) 326-6200 • Facsimile: (650) 859-2844

Abstract

When model checking concurrent software applications, symmetry reduction techniques narrow dramatically the size of the state space search by identifying computations that, because of symmetries in the system, are redundant. While state-exploration algorithms exploiting symmetry reduction are well developed, little has been done in *discovering* the nature of the symmetries of a system. What is even less researched is discovering symmetries that are particular to a temporal property. This paper proposes a general framework for discovering symmetries in systems that exhibit *absolute* or *relative* symmetries depending on the property of interest. Our work extends previous symmetry reduction techniques by making advances in automating generalized model automorphism discovery.

Contents

1 Introduction **3**

2 Motivating Application **6**

3 Defining Symmetry **9**

4 Generalized Invariance Groups **11**

5 Discovering Symmetries **12**
5.1 Property-Based Permutations 13
5.2 Exploiting Dependencies 13
5.3 Computing Dependencies 14
5.4 Refining Symmetries 15
5.5 An Algorithm for Computing Symmetries 17

6 implementation **18**

7 Discussion **20**

1 Introduction

For over half a century, formal verification has been used to help show that computer systems will behave as they are intended to. The most popular verification technique being used today is model checking, where, given a mathematical model M of the computing system and a logical formula φ that expresses a desired property of the system, the truth value of φ in M is determined by exhaustive exploration of the state space of M. The state spaces of complex systems are many orders of magnitude larger than automated model checking tools can handle. Therefore, the mathematical models of computing systems used in model checking must be much simpler than the systems that are modeled. Yet, the models must contain all the detail that is essential to the analysis being performed, because omission of relevant detail can invalidate the results of the analysis. The only obvious way of ensuring that analysis of these simple models will produce accurate results is to generate them by *abstraction*, the process of eliminating detail that has been shown to be irrelevant from more detailed models that are more obviously accurate and faithful.

Symmetry reduction is an abstraction technique that reduces the size of the state space search by identifying computations that, because of the symmetry in the model, are redundant. Models of distributed systems comprising many identical components exhibit considerable symmetry. Since multiple components play the same functional role, and therefore interact with their environment in the same way, the identity of a component during analysis is often irrelevant.

At least two natural notions of symmetry — what might be called *absolute* and *relative* symmetry — can be explicated in terms of frame automorphisms, which are, roughly speaking, one-to-one homomorphisms that map the frame $\langle S, R\rangle$ of a model M that represents a set S of states and a transition relation R, onto itself. This allows the construction of a quotient M_G formed by the construction of equivalence classes among states that are not distinguished by elements of the group G of permutations. The size of the model M_G can potentially be exponentially smaller than the size of M. The more symmetries are captured in G, the smaller M_G is. Since M_G is bisimilar to M, then for a property φ, checking whether M_G satisfies φ is equivalent to checking whether M satisfies φ.

The symmetries that can be exploited during model checking are essentially the intersection of the symmetries exhibited by the system and the symmetries exhibited by the property. These sources of symmetries characterize much of the large body of literature that studies symmetry reduction

in model checking [3, 8, 16, 5, 4, 10, 9, 17, 8]. The original work of Emerson and Sistla [9] focuses on systems with global symmetries where all components are *permutable* and where the permutations of process identity preserve the valuation of atomic predicates in the property of interest. In [5], a syntactic restriction is imposed on the description language of the system, so that the permutations of components in the system are trivially determined, but also requires a restriction on the syntax of the properties as well. Recent work [10, 17, 8] tackles systems with little or no symmetry at all. In this case, the construction of M_G is done by first computing an approximation of M_G that assumes global symmetry, and then to refine it until only the relevant symmetries that guarantee bisimulation for a given property are considered. In [17], model automorphisms are extended to more than just permutations of process identity, but also to local variable-value pairs. This more general notion is better captured by the notion of generalized frame automorphism.

While state-exploration algorithms exploiting symmetry reduction by exploring M_G instead of M are well developed, little has been done in *discovering* the symmetries of a system, and automating the construction of M_G using those symmetries. What is even less researched is discovering symmetries that are particular to a temporal property. If we can automate and speed up the process of discovering the symmetries that are possible for every single property, we would achieve better state-space reduction and provide a much more flexible framework for the analysis of a large class of distributed systems. What is of interest to us is not to exploit only symmetries that induce a bisimulation of the state graph so that all properties are preserved, but to provide an automated way of discovering some or all of the symmetries that can be exploited for every single property of interest achieving therefore better reduction than classical symmetry reduction methods and achieving various degrees of state-space reduction.

This paper proposes a general framework for discovering symmetries in systems that exhibit global or local symmetries depending on the property of interest. Our work extends previous symmetry reduction techniques by making significant advances in automating generalized model automorphism discovery. Generalized model automorphism allows us to discover symmetries between states satisfying arbitrary predicates while preserving the property of interest. Therefore, we do not only discover symmetries in systems composed of several identical processes, but we can discover symmetries in the state space of a single process. Even when processes are not symmetric and their identities can not be permuted, part of their local state space can be permuted. The contribution of this paper can be summarized

as follows:

- An anytime symbolic algorithm for discovering symmetries in systems that exhibit global, local symmetries, or no apparent symmetries at all.
- Generalization of symmetry reduction to model automorphisms that preserve a particular temporal property.
- Our schema for discovering symmetries is both incremental and conservative. At any time, the set of already computed symmetries is guaranteed to define a generalized model automorphism that guarantees the preservation of the property. This guarantees that no spurious counter-example will be generated.
- The techniques developed have been applied to a real life system that represents the largest implementation of a distributed multiagent system up to this date, and help improve its architecture. As a result of our analysis, flaws in the architecture of the system could be identified, and a set of test cases has been generated from the model checking of an abstraction of the systems that exploits symmetries. Furthermore, changes in the architecture have been suggested so that relative symmetries can be extended into absolute symmetries, making the analysis simpler and the construction of an argument of the correctness of the architecture easier.

In the larger context of automated abstraction, our approach represents a fundamentally different alternative to current automated abstraction approaches based on abstract interpretation techniques [7], where an initial coarse abstract domain is successively refined until the property is proved correct or a counter example is exhibited. This is the case for predicate abstraction [12] where an initial partition of the state space using an initial set of predicates is refined by adding more predicates when necessary [15] and therefore obtaining a more precise description of the state space. Our approach can be viewed as a bottom-up abstraction methodology that consists of first considering the systems itself and collapsing equivalent states without loss of information. The advantage is that there is no need for refinement since there is no over-approximation of the transition relation that might lead to spurious counter examples. Our philosophy consists of achieving enough state space reduction in order to be able to model check the system rather than drastically reducing the state space by means of over-abstraction and then refinement.

In the rest of the paper, we first introduce our motivating application in Section 2. In section 3, we define the notion of symmetry and model automorphism. In section 4, we describe a generalization of model automorphism. In section 5 we describe our technique for discovering generalized model automorphisms for a given property. In section 2, we describe our algorithm for discovering symmetries, and in Section 6, we discuss implementation and results of the analysis.

2 Motivating Application

Our motivating example is UltraLog, a complex distributed multiagent system that represents the largest implementation of agent system known today. UltraLog represents a challenge to known formal verification techniques in general and model checking techniques in particular. UltraLog [2] is a research project focused on creating survivable large-scale distributed agent systems capable of operating effectively in very chaotic environments. The project is pursuing the development of technologies to enhance the security, robustness, and scalability of large-scale, distributed agent-based systems operating in chaotic environments and extreme circumstances. UltraLog is built using the Cougaar technology [1]. Cougaar is a Java-based architecture for the construction of large-scale distributed agent-based applications. The UltraLog program focuses on research, development, evaluation, and demonstration of a prototype, distributed society of more than 1000 medium complexity agents. Figure 1 illustrates the architecture of the UltraLog system. Agents in UltraLog are organized in enclaves that are under different administrative supervision. Enclaves are related by wide area network connections, and every enclave includes a set of administrative agents that are used to enforce security and robustness policies. The mission of the UltraLog system requires that most agents are available when they are solicited. To ensure such a fundamental property, UltraLog relies on the following robustness mechanisms:

- *Intelligent mobility*: agents are moved to machines where the necessary resources required to perform their tasks are guaranteed to be available.
- *Failure recovery*: failure of individual agents or machines is dealt with by restarting agents on different machines using the following mechanisms:
  - automatic restart when agents fail

– migration of an agent when the machine on which it runs fails
– restoration of the state of the agents after failure using local and replicated storage

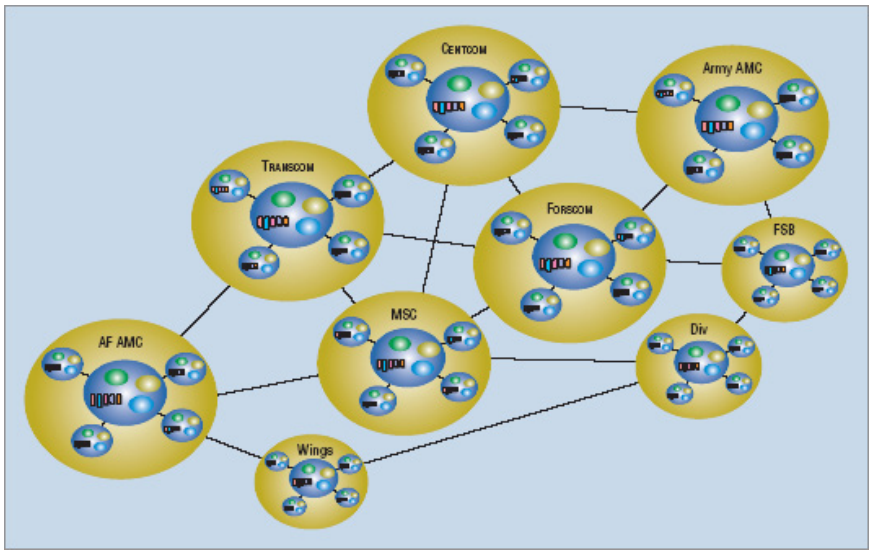

Figure 1: The UltraLog System architecture. Large ovals represent enclaves of agents, mid-sized ovals represent network nodes, and small ovals represent agents.

Every set of agents in an enclave is monitored by a set of management agents, also called *managers*. The role of a manager is to monitor a set of agents by checking that every single one of them emits regular heartbeats. When an agent's heartbeat is not received, the manager issues a ping to query the agent. When the agent does not respond to the ping, the agent is considered in a dead state and is restarted by the manager. In some instances, the security monitoring mechanisms flag the death of some agent as suspicious, and as a result might consider the whole node as potentially compromised. Therefore, the agent is restarted on a different node along with all the agents that were on the same node. A snapshot of the state of active agents is stored in a replicated storage structure regularly. Each agent, manager, and node requires an authentication certificate that enables it to interact with its environment. Authentication certificates are delivered by a set of certificate authorities (CAs) organized in a hierarchical manner.

CAs are themselves implemented as agents with the appropriate plug-ins to play their role as CAs, and they therefore require other CAs. There is one root certificate authority. When a node, a manager, an agent, or a CA is restarted, it combines a built-in certificate with the certificate of the manager that initiated the restart into a single request for a new certificate from its CA.

Our model of UltraLog is an abstraction of the current implementation in which timing and network information is abstracted away. The model is parametrized by the number of agents, robustness managers, security managers, certificate authorities, nodes, replicated storage locations, and enclaves. For simplicity, an agent, a manager, or a certificate authority can be in either its *init*, *alive*, or *dead* state , and a node can be either *up* or *down*. The global state is the composition of the states of all agents, managers, certificate authorities, their respective locations, and the state of every node.

Many aspects of the UltraLog architecture break the symmetry that one might exploit among agents, managers, CAs, or nodes. Agents, for instance, need to be distinguished depending on their managers, their CAs, and their locations, and similarly for managers, and CAs. Furthermore, depending on the property, more symmetries can be exploited. Consider, for instance, the property stating that every agent playing the role of a certificate authority that dies is eventually restarted and becomes available:

$$\varphi = \forall(i : CA) : \Box(\Diamond(alive(i)))$$

Each CA i is initially an agent in its *init* state. If i's CA defined by the predicate $certificate(i)$ is running, that is, it is in its *alive* state, then it requests a certificate from it. Once the certificate is delivered, the agent i moves to its *alive* state and starts playing its role as a CA. i can move to state *dead* in two cases: if the node on which it runs goes from its state *up* to *down*, or if i's manager is no longer receiving i's heartbeat. In either case, i is declared dead and its certificate revoked. i is then moved to another node if its original node is compromised, or restarted on the same node otherwise. Nodes on which agents run can go down in a nondeterministic way, and are restarted in a nondeterministic way as well.

Let us assume for a moment that there are three CAs such that CA 1 is the root certificate, and plays the role of certificate authority for CA 2 and CA 3. It is clear from the description of the transition relation of each CA, that the three CAs are not symmetric, and that the permutation of CA 1, CA 2, and CA 3 does not induce an automorphism of the model. So, instead of

assuming that all CAs are symmetric or *permutable* until proven otherwise following the approach of [17], we start by considering all CAs as being nonpermutable, and discovering incrementally what possible permutations they might allow. Furthermore, it is possible that even if no permutation among the CAs is possible, there exists some predicate $p(i)$ such that any state where $p(i)$ holds is symmetric to a state where $p(j)$ holds where i and j are different CAs. In fact, we can find arbitrary pairs of predicates p and q that may or may not refer to component identity, or to local variables of components so that the permutation of a state satisfying p by a state satisfying q defines a model automorphism. Using our approach, we are able to discover that CA 2 and CA 3 are permutable. Our approach determines that since CA 2 and CA 3 have the same dependencies up to permutation, they are themselves permutable. The justification is in the fact that both CA 2 and CA 3 depend on CA 1, and on their local nodes on which they run. Since the state of individual nodes depend only on the environment, node identities are permutable as well. Therefore, by permuting CA 2 and CA 3 as well as their node identities, we obtain a model automorphism. Another important condition that allows the permutation of CA 2 and CA 3 is that permuting 2 and 3 in φ leaves the property unchanged.

As a general principle for building an abstraction M_G of a model M, it is easier and more practical to start by considering M, that is, distinguish all the states of M, and then discovering states that can be collapsed, since it is not necessary to distinguish them during model checking, rather than computing a coarse abstraction and gradually refining it. It is indeed not guaranteed that the refinement is ever going to reach a converging point for an infinite state system, and might be unmanageable for systems with thousands of components, whereas with our incremental approach, any computed permutations can be used. Next, we will show how these symmetries are discovered automatically from the systems description.

3 Defining Symmetry

Before describing our procedure for discovering symmetries, we recall some preliminary definitions and define symmetry reduction. The state space of a concurrent system is represented by a transition system. Let us say that a transition system M is a quadruple

$$\langle S, R, L, P \rangle$$

where

- S is a non-empty set, called the set of *states* of M
- R is the transition relation of M
- P is a set of atomic predicates
- $L : S \to 2^P$ is a labeling function

We denote by $L(s)$ the values of the predicates of P in s. We also denote by $L(s)_{|Q}$ the values in the state s of the predicates of a subset Q of P with the convention that $L(s)_{|\emptyset} = \emptyset$. A *permutation* on a state S is a bijection of S into itself. We designate by $\mathcal{P}$ The group of permutations of S.

Definition 1 (symmetry Group) *Given a transition system $M = \langle S, R, L, P\rangle$, a subgroup G of $\mathcal{P}$ is called a symmetry group of M if for all permutations σ of G:*

$$(s_1, s_2) \in R \text{ iff } (\sigma(s_1), \sigma(s_2)) \in R$$

That is, G preserves the transition relation R. Not all permutations are property preserving. Only permutations that preserve certain atomic predicates — that is, permutations that form an invariance group — are useful.

Definition 2 (Invariance Group) *Given a transition system $M = \langle S, R, L, P\rangle$, and let Q be a subset of the set of predicates P. A symmetry group G of M is called an invariance group of M for Q if for all permutations σ of G and for every atomic proposition β in Q, $L(s)_{|\{\beta\}} = L(\sigma(s))_{|\{\beta\}}$. In other word:*

$$L(s)_{|Q} = L(\sigma(s))_{|Q}$$

That is, G preserves the transition relation R and the atomic predicates of Q. The symmetry group G defines an equivalence relation on S where the equivalence class of a state s is designated by its representative $[s]$ defined by $[s] = \{s' \in S | \exists \sigma \in G, \sigma(s) = s'\}$. This equivalence defined a quotient of the transition system M.

Definition 3 (Quotient Transition System) *Given a transition system $M = \langle S, R, L, P\rangle$, and an invariance group G of M for P, a quotient transition system for M modulo G is the transition system $M_G = \langle S_G, R_G, L_G, P\rangle$ where*

1. $S_G = \{[s] | s \in S\}$
2. $R_G = \{([s_1], [s_2]) | (s_1, s_2) \in R\}$

3. $L_G : S_G \to 2^P$ *is such that* $\forall s \in S : L_G([s]) = L(s)$

Theorem 1 *Given a transition system* $M = \langle S, R, L, P\rangle$ *and* G *an invariance group of* M *for* P*, then for any temporal property* φ *where only atomic predicates of* P *appear in* φ*:*

$$M \models \varphi \;\Leftrightarrow\; M_G \models \varphi$$

The transition systems M and M_G are bisimilar. Therefor, G strongly preserves every temporal property φ with atomic predicates in P.

4 Generalized Invariance Groups

Invariance groups capture the symmetries in a transition system and allow the preservation of all properties that use only atomic predicates in P. This restriction can be removed by allowing that symmetric states s and $\sigma(s)$ do not necessarily agree on the predicates of P.

Definition 4 (Generalized Invariance Group) *Given a transition system* $M = \langle S, R, L, P\rangle$*, a symmetry group* G *of* M*, and a mapping* $\gamma : P \to P$*, the pair* $\langle G, \gamma\rangle$*, denoted by* G_γ*, is called a generalized invariance group of* M *for all permutations* σ *of* G*:*

$$\forall s \in S, L(s) = \gamma(L(\sigma(s)))$$

The generalized invariance group defines a generalized automorphism of M where equivalent states are not required to agree on the set of atomic predicates. This generalization that admits arbitrary swapping of atomic predicate will achieve more symmetry reduction for some properties. The most trivial ones are the properties φ such that φ and $\gamma(\varphi)$ are equivalent, where $\gamma(\varphi)$ is defined as the property φ where every atomic property β appearing in φ is substituted by $\gamma(\beta)$.

Considering generalized invariance groups is a powerful approach to state space reduction. Permutations between states are not restricted to permutation of process indices anymore. Even if two processes i and j are not symmetric, some of their respective behaviors might be. Therefore it is possible that two predicates $p(i)$ and $p(j)$ can be symmetric but not i and j. Another consequence of considering generalized invariance groups is that symmetry reduction can not be restricted to concurrent systems defined by the composition of identical processes, but can be defined in general for arbitrary systems.

Theorem 2 *Given a transition system $M = \langle S, R, L, P\rangle$, a generalized invariance group G_γ of M for P, and any temporal property φ where only atomic predicates of P appear in φ. If φ is invariant under γ, that is, $\varphi = \gamma(\varphi)$ then:*

$$M \models \varphi \;\Leftrightarrow\; M_{G_\gamma} \models \gamma(\varphi)$$

From now on, we consider φ to be an LTL temporal formula. The main importance of G_γ and similar sorts of structure-preserving mappings is that a computation

$$\tau = \langle s_0, s_1, s_2, \ldots\rangle$$

and its image under σ

$$\sigma(\tau) = \langle \sigma(s_0), \sigma(s_1), \sigma(s_2), \ldots\rangle$$

cannot be distinguished by a certain class of logical formulas. Hence, there is no need for considering both τ and $\sigma(\tau)$ when searching for a refutation of φ. In other words, after a (generalized) model automorphism has been discovered, each state s can be identified with its image $\sigma(s)$ under σ to reduce the size of the state space prior to model checking. One obvious potential source of model automorphisms in agent-based systems is swaps of agents — or groups of agents, as in the UltraLog application agent/robustness manager/certificate authority example sketched earlier — that play the same role in a system and that do not interact with one another. The main source of generalized model automorphisms is swaps of states of individual agents with the same agent states as predecessors and successors. (Not every such swap is of interest as there is, of course, no guarantee that the LTL formula which expresses the property to be checked will be invariant under the permutation of atomic formulas that corresponds to the swap.) Although the basic idea of using symmetries to simplify model checking is well known, we have extended previous symmetry-reduction techniques such as the ones in described Sistla and Godefroid's paper [17] in ways that make them more generally applicable to the sort of systems that we are interested in. In particular, we have made significant advances in automating generalized automorphism discovery. During the presentation, we will omit some of the obvious proofs.

5 Discovering Symmetries

We propose an algorithm that takes as input system description and a temporal property φ and produces a generalized invariance group G_γ. Since G_γ

is computed so that it preserves only the property φ, dramatic state space reduction can be achieved. For a given property φ, our approach consists of finding a pair predicates (p_i, q_i) that defines a generalized automorphism such that σ_i is defined as the mapping between states satisfying p_i and states satisfying q_i. Once such pair is found, it is possible to find new pair (p_j, q_j) of predicates such that its corresponding σ_j defines also a generalized automorphism. The composition of σ_i and σ_j is therefore also a generalized automorphism. We first focus on permutations of states that form a generalized automorphism and then find the predicates that characterize the sets that can be safely permuted for the purpose of reducing the explored state space during model checking.

5.1 Property-Based Permutations

Definition 4 is our starting point in finding symmetries. Let us first define permutations of states that preserves a given property φ.

Definition 5 (φ-Permutation) *Let φ be an LTL formula. A mapping $\sigma : S \to S$ is a φ-permutation if for every infinite sequence τ*

$$\tau \models \varphi \quad iff \quad \sigma(\tau) \models \varphi$$

By definition, φ-permutations preserve the property φ. We propose an algorithm for computing such permutation by showing that they satisfy, by construction, condition 2 of Definition 4. That is, they preserve the property φ. We will then show that for every infinite sequence of states τ of a system M, the permutation $\sigma(\tau)$ is also an infinite sequence of M. That is, condition 1 of Definition 4 is satisfied.

Given a model M and a temporal logic property φ, the problem of discovering symmetry is the problem of finding a set of φ-permutations. We propose an algorithm for finding permutations by computing and then successively refining an initial permutation. Our incremental approach allows us to provide users with a tradeoff between the amount of reduction they want to achieve and the amount of symbolic computation that is required to generate the permutations.

5.2 Exploiting Dependencies

One obvious way of discovering φ-permutations is to exploit dependencies. Intuitively, for every sequence of states, any state that does not influence the property φ can be substituted by another state that also does not influence the validity of φ.

Definition 6 (dependency graph) *Let p be a predicate. The dependency graph dep_graph(p) for p is an abstract transition graph where every state is a predicates and there exist an edge from predicate p to a predicate q if the validity of p depends on q. In other words, for every state s satisfying p, there is a state s' satisfying q such that $(s', s) \in R$.*

The dependency graph for φ is constructed by tracking the dependencies of every atomic predicate in φ. It can be computed in many different ways such as by a simple backward computation by collecting all atomic predicates appearing in the transitive closure of the precondition of every atomic predicate in φ. The dependency graph is potentially infinite, however, for the purpose of discovering symmetries, it is not necessary to compute the whole graph as we will see later. But for now, let us assume that the graph is a finite graph, and let $dep(\varphi)$ be the set of all predicates appearing in the graph.

Proposition 1 *Let s and t be two states such that $L(s)_{|dep(\varphi)} = L(t)_{|dep(\varphi)}$. The permutation σ that maps s to t and vice versa, and maps every other state to itself, is a φ-permutation.*

Computing a first set of dependencies can be used to slice a model, and therefore reduce the state space. This can be expressed as a permutation σ_0 defined as follows:

$$\forall s \in S \quad \sigma_0(s) = \begin{cases} t & \text{if } L(s)_{|dep(\varphi)} = L(t)_{|dep(\varphi)} \\ s & \text{otherwise} \end{cases}$$

That is, σ_0 does not distinguish states that agree on the valuation of predicates in $dep(\varphi)$. We consider σ_0 to be the first permutation to be extracted from the property φ.

5.3 Computing Dependencies

As stated previously, the dependency graph for φ is constructed by tracking the dependencies of every atomic predicate in φ. It can be computed in many different ways such as by a simple backward computation by collecting all atomic predicates appearing in the transitive closure of the precondition of every atomic predicate in φ. The construction is done using the program description given as a set of guarded commands.

Definition 7 (Program) *A program $\mathcal{P}$ is a tuple $\mathcal{P} = \ < \mathcal{V}, \mathcal{T} = \{\tau_1, \cdots, \tau_n\}, Init >$, where*

- $\mathcal{V}$ *is a set of program variables.*
- $\mathcal{T}$ *is a set of transitions or guarded commands.*
- *Init is a predicate characterizing the set of initial states.*

Each transition τ is a guarded command

$$guard \longrightarrow v_1 ::= e_1, \cdots, v_n ::= e_n$$

where $\{v_1, \cdots, v_k\} \subseteq \mathcal{V}$. The boolean expression *guard* is the guard of the transition τ. Each variable v_i is assigned with an expression e_i of a compatible type. A state of a program $\mathcal{P}$ is a valuation of the system variables of $\mathcal{V}$. We also recall the definitions of predicate transformers over transition systems. The predicate transformer *pre* expressing the precondition by a transition τ of a predicate p over the state variables of $\mathcal{V}$ are defined as follows:

$$pre[\tau](p) \ = \ \exists \mathcal{V}'.action_\tau(\mathcal{V}, \mathcal{V}') \wedge p(\mathcal{V}')$$

where $action_\tau(\mathcal{V}, \mathcal{V}')$ is defined as the relation between the current state and next state, that is, the expression

$$guard \wedge \bigwedge_{i=1}^{k} v_i' = e_i$$

The semantics of a program P is given by its computational model M represented by a transition system defined in Section 3.

Computing the dependency graph for any predicate p uses the *pre* operator as follows: The root of the graph is the predicate p it self. For each predicate q in the conjuncts of $pre[\mathcal{P}](p)$, an arc between q and p is added.

5.4 Refining Symmetries

Exploiting dependencies allows the permutation between sequences of states that do not affect the property φ. While this can sometime lead to a significant reduction in the state space to explore, it basically exploits a very simple case of symmetry. We propose to further exploit dependencies by refining the already computed symmetries and discovering more symmetric states. Given a φ-permutation σ_0 such as the one computed previously, we know that σ_0 does not distinguish states that agree on the valuation of predicates in $dep(\varphi)$. Our approach for refining σ consists in finding a pair of predicates p and q in $dep(\varphi)$, such that the permutation between

states that agree on all the valuations of predicates in $dep(\varphi)$ but p and q is a φ-permutation. That is, the permutation of states s_1 and s_2 satisfying respectively p and q is a φ-permutation.

Definition 8 *Let p and q be two predicates in $dep(\varphi)$. p and q are permutable if $\forall p' \in dep(p), \forall q' \in dep(q)$ one of the following is true:*

- *p' and q' are equivalent.*
- *p' and q' are permutable.*

In other words, states in p and q can be permuted if and only if their dependencies are the same up to symmetry. That is, they depend on the same states, or they depend on states that are themselves permutable. Here we realize that when computing the dependency graphs for p and q, it is enough to compute just enough dependencies to either show that those dependencies are equal or permutable. This is useful when the system under consideration is infinite and the dependency graph can be potentially infinite. While this definition allows us to define what states are permutable, the following proposition provides a way of finding p and q that are relevant to a particular property.

Proposition 2 *Let s and t be two states such that s and t satisfy respectively the atomic predicates p and q of $dep(\varphi)$. Let σ be the permutation such that $\sigma(s) = t$, and $\sigma(s') = s'$ for every state s' satisfying neither p nor q. The permutation σ is a φ-permutation if and only if both of the following are true:*

- *p and q are permutable*
- $\forall(s \in p) : \forall(t \in q) : \varphi[s/t] = \varphi$

Proof: The proof is by induction on the structure of the dependency graphs of p and q.
□

This proposition allows us to recursively construct a set of φ-permutations starting from an initial φ-permutation defined by exploiting dependencies. This allows us to discover the symmetries in a system in an incremental way. The refinement of the initial permutation σ_0 by identifying two predicates p

and q that are permutable from the set of predicates $dep(\varphi)$ can be expressed by the following permutation:

$$\sigma_{(p,q)} = \begin{cases} \sigma_{(s,t)} & \text{if } p = q \\ \sigma_{(s,t)} \circ \sigma_{p',q')} & \text{if } \forall p' \in dep(p), \forall q' \in dep(q),\ p \text{ and } q \text{ are permutable} \\ \text{identity otherwise} \end{cases}$$

where $\sigma_{(s,t)}$ is defined as follows: $\forall s \in p \ \ \sigma_{(s,t)}(s) = t \in q$

5.5 An Algorithm for Computing Symmetries

The initial permutation and its successive refinements can be combined to form a single permutation that can be applied to the entire state space. The following proposition shows that the composition of all permutations computed by our algorithm is a permutation that allows symmetry reduction.

Proposition 3 *Let σ_i be a φ-permutation and let σ_j be a φ-permutation. The composition $\sigma_i \circ \sigma_j$ is also a φ-permutation.*

Our algorithm for computing permutations is described in Figure 2. We

Begin
$\mathcal{X} = \mathcal{X}_0 \cup pred(\varphi)$;
$\sigma = \sigma_0$;
while $\mathcal{X} \neq \emptyset$ **Do**
chose q $\in$ $\mathcal{X}$ and p $\in$ $\mathcal{X}$ such that $p \cap q = \emptyset$;
compute $dep\_graph(p)$;
compute $dep\_graph(q)$;
if $permutable?(p,q) \ \wedge \ \forall(s \in p) : \forall(t \in q) : \ \varphi[s/t] = \varphi$
then
$\sigma = \sigma \circ \sigma_{(p,q)}$;
$\mathcal{X} = \mathcal{X} \ \setminus \ \{p\} \cup \{q\} \cup dep(p) \cup dep(q)$;
endif
Od
End

Figure 2: Algorithm for computing symmetries

consider an initial arbitrary set $\mathcal{X}$ of candidate predicates that includes the predicates in the property φ. We pick two predicates p and q and compute their dependency graphs. If p and q are permutable, then p and q and tare removed from $\mathcal{X}$. We also remove the predicates appearing in

the dependency graphs from $\mathcal{X}$ since if two permutable predicates p' and q' appear in $\mathcal{X}$ and also appear in the dependency graphs of p and q, then the permutation $\sigma_{(p,q)}$ subsumes the permutation $\sigma_{(p',q')}$.

Theorem 3 *Let σ a permutation computed by the algorithm of Figure 2 for a property φ. The permutation σ is a φ-permutation, and for every infinite sequence τ of the corresponding transition system M, $\sigma(\tau)$ is also a sequence of M.*

Proof: First, we can establish that for a given p and q, $\sigma_{(p,q)}$ is a φ-permutation. We can also establish that the composition of two φ-permutations is a φ-permutation. Finally, the definition of dependency graphs allows us to show that $\sigma_{(p,q)}$ preserves the transition relation R.
□

6 implementation

UltraLog is modeled using the SAL specification and verification environment[1]. SAL stands for Symbolic Analysis Laboratory. It is a framework for combining different tools for abstraction, program analysis, theorem proving, and model checking toward the calculation of properties (symbolic analysis) of transition systems. A key part of the SAL framework is an intermediate language for describing transition systems. This language serves as the target for translators that extract the transition system description for popular programming languages such as Java, and Statecharts. The intermediate language also serves as a common source for driving different analysis tools through translators from the intermediate language to the input format for the tools, and from the output of these tools back to the SAL intermediate language. Our dependency analysis uses the SAL slicer developed in [11]. The input of the slicing algorithm consists of the slicing criterion and a SAL description of the system in the form of the parallel composition of modules. The slicing criterion is in our case a set of local and global variables appearing in the set of atomic predicates in the formula. The output of the slicing algorithm is another SAL description of the same system wherein irrelevant code from each module has been sliced out. When considering two predicates p and q, the slices with respect of the variables appearing p and the variables appearing in q are compared for equivalence or for permutability.

[1] http://sal.csl.sri.com

As a result of using the slicing program as a dependency analysis tool, we can guaranty that the permutations generated by our algorithm correspond to runs in the system.

The following is the description of the results of the algorithm for two different properties. The first property expresses that every agent playing the role of a certificate authority that dies is eventually restarted and becomes available:

$$\phi = \forall(i : CA\_id) : \Box(\Diamond alive(i))$$

The dependency computation allows us to generate the following predicates that appear in the SAL description of UltraLog:

- $location(i) = n$
- $location(ca^*(i)) = n$
- $ca^*(i) = c$

These predicates refer to the free variables n and c representing respectively nodes and certificate authorities. These predicates indicate that in order to verify the property above, it is not necessary to consider the behavior of any agent nor any manager, but only the certificate authorities and their associated certificate authorities, and the nodes they run on. This allows us on one the one hand to consider all possible permutations among agents as well as among all managers, and on the other hand to exploit a symmetry among certificate authorities. Since $ca(i)$ designate the certificate authority of agent i, we realize that if i and j are two CAs agents, then i and j are permutable if they have the same CA. A similar property can be verified for application agents.

$$\forall(i : Agent\_id) : \Box(\Diamond alive(i))$$

The analysis of the dependencies of the predicate $alive(i)$ leads to the following predicates, and their corresponding permutations.

- $location(i) = n$
- $\exists j : Agent.\ location(i) = location(j) \wedge manager(j) = manager(i)$
- $location(ca^*(i)) = n$
- $location(manager(i)) = n$
- $location(ca^*(manager(i))) = n$

- $enclave(n) = enclave(location(i))$
- $ca^*(i) = c$
- $ca^*(manager(i)) = c$

That is, agents i and j are permutable if and only if, their corresponding CAs and managers are permutable, and if their locations are permutable. More details about the example can be found at

`http://www.csl.sri.com/users/saidi/symmetry/`.

7 Discussion

We have presented and incremental approach for finding symmetries by automatic analysis of the description of a distributed system given in a guarded command language such as SAL. The approach is applicable to a large number of systems and is not restricted to systems composed of identical processes. The approach is based on finding predicates that are permutable. Our approach solves the problem of finding the symmetries in order to apply symmetry reduction techniques, where the symmetries are given by the user or are trivial due to restrictions on the syntax of the description of the system and the properties. Our approach can be seen as a composition of symmetry reduction and predicate abstraction [12]. It is possible to prove that the generated predicates can be used to build a predicate abstraction of the original system that preserves the property of interest. The choice of candidate predicates can either be guided by the user or by heuristics such as the ones defined in [14] for instance. The reduced model for various properties have used for the purpose of test case generation based on the techniques described in [13]. The test cases have been useful in identifying flaws in the UltraLog implementation and architecture.

References

[1] Cognitive agent architecture (cougaar) open source project. www.cougaar.org.

[2] Ultralog: A defense advanced research projects agency program on logistics information system survivability. www.ultralog.net.

[3] K. Ajami, S. Haddad, and J.-M. Ilié. Exploiting symmetry in linear time temporal logic model checking: One step beyond. *Lecture Notes in Computer Science*, 1384:52–62, 1998.

[4] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In Courcoubetis [6], pages 450–462.

[5] C.N. Ip and D.L. Dill. Verifying systems with replicated components in murphi. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 147–158, New Brunswick, NJ, USA, July/Aug. 1996. Springer Verlag.

[6] C. Courcoubetis, editor. *Fifth International Conference on Computer-Aided Verification*, volume 697 of *LNCS*, 1993.

[7] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. pages 238–252. ACM Press, Jan. 1977.

[8] E. Emerson, J. Havlicek, and R. Trefler. Virtual symmetry reduction. In *15th Symposium on Logic in Computer Science (LICS' 00)*, pages 121–131, Washington - Brussels - Tokyo, June 2000. IEEE.

[9] E. A. Emerson and A. P. Sistla. Symmetry and model checking. In Courcoubetis [6], pages 463–478.

[10] E. A. Emerson and R. J. Trefler. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In *Proceedings of the Conference on Correct Hardware Design and Verification Methods*, volume 1703 of *lncs*, pages 142–156, 1999.

[11] V. Ganesh, N. Shankar, and H. Saïdi. Slicing SAL. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA 94025, 1999.

[12] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. volume 1254, pages 72–83, Haifa, Israel, June 1997.

[13] G. Hamon, L. deMoura, and J. Rushby. Generating efficient test sets with a model checker. In *2nd International Conference on Software Engineering and Formal Methods*, pages 261–270, Beijing, China, Sept. 2004. IEEE Computer Society.

[14] K. S. Namjoshi and R. P. Kurshan. Syntactic program transformation for automatic abstraction. In *Proceedings of the International Conference on Computer Aided Verification CAV*, volume 1855 of *LNCS*, pages 435–449, 2000.

[15] H. Saïdi. Model-checking guided abstraction and analysis. In *7th International Static Analysis Symposium, SAS 2000*, volume 1824, pages 377–396, June 2000.

[16] Sistla, Gyuris, and Emerson. SMC: A symmetry-based model checker for verification of safety and liveness properties. *ACMTSEM: ACM Transactions on Software Engineering and Methodology*, 9, 2000.

[17] A. P. Sistla and P. Gedefroid. Symmetry and reduced symmetry in model checking. In *Proceedings of the Thirteenth International Conference on Computer Aided Verification CAV*, volume 2102 of *LNCS*, pages 91–103, 2001.