The Dissertation Committee for Thomas Wahl
certifies that this is the approved version of the following dissertation:

Exploiting Replication in Automated Program Verification

Committee:

____
E. Allen Emerson, Supervisor

____
James C. Browne

____
Warren A. Hunt, Jr.

____
Vladimir Lifschitz

____
Dewayne E. Perry

Exploiting Replication in Automated Program Verification

by

Thomas Wahl, Univ.-Diploma, M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2007

Acknowledgments

Like any reasonably large scientific work, this dissertation bears the handwriting of more than one person. Some helped with their ideas, others with their critique. Some influenced me directly, others shaped my personality by their example. Reflecting back upon the years behind me, I find that I owe to all of these people.

Not because it is customary, but because it is appropriate: I first thank my advisor, Prof. Allen Emerson, for his guidance. He has helped my career in more ways than he is probably aware of. It was both an honor and—sometimes—a challenge to work under the supervision of such a distinguished member of the research community. Most valuable to me was his rigor in reviewing my work. His refusal to look at immature drafts taught me to be self-critical before imposing my work onto others. I continue to look up to the high standards of his own groundbreaking work in the early days of formal verification.

A lesson for life was Allen's focus on *truly* important things, adjourning clerical tasks with an amazing degree of stubbornness. Despite his immense repertoire of scientific accomplishments, he remained a person humble at heart who can talk to his students like a friend. I will miss our conversations.

Profs. J. Browne and W. Hunt urged me to heed the practical aspects of research. Prof. Hunt introduced me to hardware verification and helped establish contact with Intel, where I had the chance to do an internship during the late years of my Ph.D. program. Prof. Browne sharpened my vision regarding the distinction of “are we doing things right” vs. “are we doing the right things”. His contributions to my career, however, go beyond research. I suspect his calendar to be tight; nevertheless he offered advice on many occasions, such as during the job search process. I wish I could learn more from him about effective scheduling.

I am grateful for Profs. V. Lifschitz and D. Perry for their service as committee members. Incidentally, I have known Prof. Lifschitz for more than a decade, when I started to learn rigorous mathematical logic from him. No-one can seriously do research in formal methods without a solid logic background.

Prof. J. Misra introduced me to the intricacies, and beauties, of distributed systems algorithms. I also had the pleasure to work with him on language design, which allowed me insights into how to encourage correct programming. I appreciate his support during my job search, as well as his general advice on life in academia.

I specifically thank Prof. Emerson’s former students Richard Trefler and Nina Amla. I find that several of my papers were inspired in part by Richard’s research on symmetry, and I am glad that we were able to publish one result together. While Nina was particularly resourceful during my decision to choose formal methods as my research field, it was Richard’s diligence in giving advice on finishing up and on finding a job that deserves my utmost gratitude.

I had most fruitful discussions with my colleagues Fei Xie and Sandip Ray. Fei, who is now a professor at Portland State University, gave plenty of advice on life as a faculty member, and on becoming one. I was very fortunate for Sandip to decide to stay at UT after graduation, so I could enjoy discussions with him (although not often enough), benefit from his encyclopedic knowledge, and exchange jokes with him about our unequal advisors. I admit that I often peeked at these fine colleagues’ dissertations to see what it *should* be. I appreciate the company of Jyotirmoy Deshmukh (did I get that spelled right?) and Roopsha Samanta as my later fellow advisees of Prof. Emerson. Please don’t repeat my mistakes!

Last but not least, I want to thank the department's staff for taking care, professionally, of so many things that form the administrative scaffolding of a Ph.D. program. I point out Katherine Utz's cheerful disposition when assisting students, and Gloria Ramirez' friendliness, knowledge about "the procedures" and uncompromising willingness to help, even if all is explained clearly on forms 1-3.4/IVa-c. I also thank the people who sat down and did the obvious: create a dissertation template that abides by the University's format guidelines. Thanks to the template, my dissertation grew from 0 to 30 pages in under ten seconds. Seriously, their work saved generations after them much hassle.

Now that all seems done, it is tempting to say: "Actually, getting the Ph.D. was easy"—it was not. Anyone who says it was is probably lying. It is a long process. There was a time when my advisor kept asking me, "How many papers do you have?" Worse, I kept giving him the same answer. At the end they even started demolishing beloved Taylor Hall, and I had to vacate my office.

I want to conclude mollifyingly by thanking the state of Texas and its people. In all those hours when I should have been exploring state spaces, I instead explored the cute villages, the vast lands, or the infinite coastline of the state. I found friendly folks everywhere. I am truly grateful for having found a second home here, so far away from the first, and I am looking forward to coming back.

Thomas Wahl

The University of Texas at Austin
August 2007

Exploiting Replication in Automated Program Verification

Publication No. ______________________

Thomas Wahl, Ph.D.
The University of Texas at Austin, 2007

Supervisor: E. Allen Emerson

This dissertation shows how systems of many concurrent components, which naively engender intractably large state spaces, can nevertheless be successfully subject to exhaustive formal verification, provided the components can be classified into a few types. It therefore addresses an instance of the *state explosion problem*: a finite-state model of a system can be much larger than a high-level description of this system. Model checking, the technique to which this dissertation is primarily devoted, inherently relies on state space exploration and thus suffers from this problem more than other formal verification methods.

The state explosion phenomenon persists even if the system consists of components that are simply replicated instances of a generic behavioral description. Examples of such systems abound; they include processes executing concurrently according to some common protocol, clusters of processors executing a parallel program, and hardware with replicated physical devices in a uniform arrangement. Fortunately, models of such systems often exhibit a regular structure, known as *symmetry*, which can be exploited in verification, sometimes to the extent of an exponential reduction in model size.

The first contribution of this dissertation is to show how reductions based on symmetry can be performed with state-of-the-art system representations. Many of today's computing systems induce astronomically large state spaces whose formal models require a symbolic encoding using Boolean formulas. Such succinct representations call for new algorithms that can process entire sets of objects at once. How to detect symmetry quickly during symbolic model checking, so that redundancy in the exploration can be avoided, was an open problem for some time. In this dissertation we provide an efficient and flexible solution to this problem by using symbolic data structures in a somewhat unconventional way.

The second contribution is to extend symmetry reduction techniques to more realistic and general scenarios. We establish that the principal ideas still apply if symmetry is violated in parts of the state space. Such scenarios are common in practice, for instance when priorities determine which of several competing processes can access a resource first. In these situations it is beneficial to exploit symmetry where it exists and watch out for the (few) violations, rather than to ignore it altogether. We also demonstrate how symmetry can help us solve a practically significant instance of *parameterized verification* of system families. This technique traditionally attempts to prove properties about systems independently of the size parameter, but requires models of a special structure. We show that by restricting the parameter to a finite range we can solve this problem efficiently, can do so without any conditions on the models' structure, and can take advantage of symmetry in the individual systems of the family if it exists.

Contents

List of Figures

List of Algorithms

List of Tables

Part I

Introduction

Chapter 1

About This Dissertation

Systems of many concurrent components naively engender intractably large state spaces. They can nevertheless be successfully subject to exhaustive formal verification, provided the components can be classified into a few types.

1.1 Problem Overview

Reliability of computer systems affects everyone today, even those ignorant of the ubiquitous presence of computers in society. This insight has been widely accepted throughout the software and hardware industry, especially when applied to safety-critical and economically vital applications. Opinions diverge when it comes to the means of achieving reliability and range from "getting it right the first time" to after-the-fact methods such as testing, simulation and formal verification. The justification for automated formal techniques lies in the observation that due to the immense complexity of today's systems, human capacity is insufficient to produce correct programs at the first attempt, or to eliminate all errors afterwards by trial-and-error approaches. Instead, computers must be used to establish the correct functioning of computer programs. This implies the need for a formal approach that can be implemented on a machine.

Formal methods have the potential to be exhaustive, to provide guarantees, and—in some cases—to explain what has been found, by a proof or a counter example. This potential comes at a price. Users of formal tools must usually have more training than testing engineers; just how much depends on the specific technique. One that aims at reducing the required level of expertise is *model checking* [CE81, QS82]. Prerequisites for its employment are that the system at hand can be represented using a finite-state structure, and that the property of interest is expressible in a formula of a special decidable logic. Determining whether the property is satisfied by the system then amounts to deciding whether the structure satisfies the formula. The techniques in this dissertation were developed with model checking in mind. Some are, however, of a more general nature; we point out such cases in the text.

The potential of model checking to be exhaustive, i.e. to provide full coverage, is achieved by a sophisticated search through the state space. This search, however, is also responsible for the major obstacle that model checking faces in practice: the *state explosion problem*. The formal model that is needed in order to systematically explore the system is often much larger than the original system description. As a result, straightforward use of model checking can result in unsatisfiable or at least unacceptable time and memory needs.

To increase the number of states they can handle, many model checkers today avail themselves of some form of *symbolic representation*: Sets of states are expressed as Boolean constraints over the state variables. This way, a relatively short Boolean formula can often represent a number of states much larger than it would be possible to enumerate individually. Moreover, most operations important for model checking can be implemented efficiently on symbolic data structures. The ensuant idea of *symbolic model checking*, proposed in a landmark dissertation by K. McMillan [McM93], has had tremendous success since the mid 1990s and has increased the scope of model checking to some systems with extremely large state spaces.

Coping with the state explosion problem is one major thrust of model checking research today; success in this regard will likely have a crucial impact on the future of the technique. This dissertation presents solutions to this problem for a frequent and notorious type of systems: those consisting of many components that are *replications* of a generic behavioral description. The components can be abstract entities, such as processes in a concurrent system, each executing a copy of a program instantiated by a unique process identifier. The components can also be physical entities, such as memory locations access to which is regulated by a cache consistency protocol. The global state space, induced by the concurrent existence of the components, is of size exponential in the number of components: despite interactions among them, the local states of the components are largely independent, and any combination of local states is a conceivable global state. This explosion renders naive exploratory approaches infeasible.

Reason for hope comes with the observation that the aforementioned components may be interchangeable. To stay with the above example, in a concurrent system of processes running the same program, it seems intuitive that there is no *order* among the processes. That is, in a two-process system, the state where one process has access to the CPU and the other does I/O is intuitively equivalent to the state where the rôles are reversed. Consider how model checking represents a global state of such systems: as a vector of local states of processes. For example, the two states in the two-process system above may be stored as $(\texttt{CPU}, \texttt{I/O})$ and $(\texttt{I/O}, \texttt{CPU})$. This very step introduces an unnecessary and in fact undesired order among the processes: technically, the above two global states are different and will be distinguished during a state space search, despite their resemblance. This introduces redundancy into the search, which can, however, be factored out.

Systems with interchangeable, unordered components are characterized by transition models that exhibit *symmetry.* That is, for storage purposes these models do impose an order on the components, but reordering the components in certain ways results in exactly the same model: its set of transitions is invariant under such reorderings. This demonstrates that the order is an artifact of the modeling processes and should be ignored during model checking. This is done by considering states such as $(\texttt{CPU}, \texttt{I/O})$ and $(\texttt{I/O}, \texttt{CPU})$ equivalent: they are considered instances of a single *abstract* state. In general, an abstract state comprises all states identical up to reorderings of the components. This compression of equivalent states into one is known as *symmetry reduction.* If the transition set is invariant under any reordering, an abstract state may represent exponentially many original *concrete* states. In this case, symmetry reduction yields an exponentially smaller abstract system model. This model can be shown to have the same properties as the original one, as long as the properties do not artificially distinguish among the components. For example, instead of asking whether it is possible for process 1 to get into a bad state, the question should be whether *any* process can get into a bad state. Just like the original system itself, this property does not impose an order on the processes and can thus be verified over the much smaller abstract model.

In practice, it turns out to be much more difficult to exploit symmetry than the above relatively simple observations seem to suggest. In order to compress equivalent states into a single abstract state, it is necessary to recognize them as such, and this recognition should be cheap. This appeared to be impossible if the model is represented symbolically, using Boolean constraints, as is usually the case for very large systems. In this case, equivalence between states must be expressed as a Boolean constraint as well. This constraint is a Boolean formula over two states evaluating to true if the states are identical up to reorderings of the components. It turned out that such a formula essentially requires an enumeration of all possible pairs of reorderings, rendering symbolic representation meaningless. This notorious problem was investigated in depth by E. Clarke, R. Enders, T. Filkorn and S. Jha [CEFJ96] and has since become known as the *orbit problem*, after the name *orbit* for a symmetry equivalence class. The verdict was that symmetry can be combined with symbolic representation only in very rudimentary ways and that one cannot have the best of both worlds.

A second problem occurring in practice is that the mathematical principles of symmetry may not always hold precisely, but perhaps approximately. Suppose a system design assigns priorities to a set of perfectly replicated components, for example to avoid deadlocks. Priorities impose an order on the components and thus formally destroy symmetry. On the other hand, the priorities matter only in situations when several components attempt to access a resource of which too few copies are available. Most behavior of such a system is still invariant under reorderings of the components. Ignoring the special structure would clearly generate a model checking procedure that spends much of its time on redundant work.

In summary, symmetry seemed to be hard to extract from compact data structures, which are otherwise necessary to be able to represent large systems, and the mathematical theory of symmetry and symmetry reduction did not cater for practical cases of symmetry with small defects. These reasons discouraged the use of symmetry as a means to fight state explosion.

1.2 Results Overview

The constraint that governs the use of any reduction technique is that the (unavoidable) reduction overhead must not devour the reduction benefit. The general theme of this dissertation is to demonstrate how the overhead can be minimized to a level that makes exploiting symmetry worthwhile.

Any scheme for model checking under symmetry in some form accommodates the steps (i) *modeling*, (ii) *reducing* and (iii) *checking*. For example, the principal (but in practice infeasible) approach to symmetry reduction outlined in section 1.1 can be viewed as building a formal model (i), followed by deriving a reduced abstract model (ii), followed by model checking the reduced model (iii). Alternative strategies devised in this dissertation are obtained by combining the step (i)-(iii) in different ways.

Dynamic symmetry reduction. We present an approach to symbolic symmetry reduction that interleaves the reduction (ii) and the model checking process (iii). We show that the transitions of the abstract model can be faithfully simulated by applying the transitions of the unreduced model, followed by an adjustment that maps the produced states to abstract states. The reduction step is hence embedded in the model checking process; the set of abstract transitions is not needed.

We call this approach *dynamic* to discriminate it against the traditional paradigm, where the reduction is obtained statically before model checking. The dynamic approach has two advantages. The first is that the abstract system is never built and the orbit problem thus avoided—with all the savings this entails. This benefit alone makes the approach worth considering. The second advantage is that the procedure is applied to reachable states only. This provides tremendous gains if errors are found early, i.e. close to the initial state. In contrast, building the abstract system before model checking always applies to the entire state space.

Counter abstraction. Another approach to symbolic symmetry reduction is based on the observation that in some cases the reduction step (ii) can be performed before the modeling step (i), by applying the reduction on the program text. This technique had been proposed before in the form of *counter abstraction*, to allow reasoning about systems composed of many identical processes. It was later re-discovered as a means to symmetry-reducing symbolically represented systems.

This method is based on the idea that in a system of many indistinguishable components, it is sufficient to count, for each conceivable component configuration, how many components reside in it, instead of storing, for each component, its configuration. With some effort it is possible to translate the input program into one that operates over component counters. A model derived from the new program is of size polynomial in the number of components, making it attractive for systems with many identical processes. The caveat is that the model size is exponential in the number of component configurations.

In the dissertation we show techniques that alleviate the unfavorable dependence of counter abstraction on the number of component configurations. An over-approximation of the input program can be used to estimate the set of reachable configurations; the counters for unreachable ones are known to be zero and can be omitted. In addition, program variables whose value is guaranteed to be irrelevant in the future at certain program lines can be collapsed in configurations. All these techniques are performed on the program text, before modeling, and thus help reduce state explosion before it happens.

Adaptive symmetry reduction. The dissertation also presents a technique to explore the state space of systems that are not formally symmetric. The technique is based on the idea that—as long as the system is *approximately* symmetric—more compression can be achieved by assuming the system was indeed symmetric and record exceptions on the fly, than it is to ignore the existing symmetry altogether. Exceptions are detected from the program text; they often take the form of guards that allow only certain processes to perform certain actions, for example only processes with high enough priority to access a shared resource. Exceptions to perfect symmetry cause explored states to be annotated in a way that impacts future exploration from the state; in this sense the algorithm *adapts* to encountered symmetry violations.

Symmetry and parameterized reasoning. Finally, this dissertation establishes a connection between two related techniques that both reason about systems with replicated components: The objective of *parameterized verification* is to identify classes of systems for which it can be proved that a certain property holds independently of the system size, i.e. the number of components. Such proofs often have the form that the property is true for any size exactly if it is true for all systems up to some bound. This reduces a problem over an infinite system family to one over a finite family. In the dissertation we describe how the problem over a finite family can in turn be reduced to one over a single system, which we call the *aggregate* of the systems in the finite family. We also show that if the parameterized systems in the family exhibit symmetry, then so does the aggregate system, and symmetry reduction can be applied to it. This shows that the proof obligation posed by parameterized reasoning can be discharged efficiently.

This technique has an important ramification beyond symmetry. If the individual systems in the family are *heterogeneous*, i.e. a given property is true for some systems but not for all, then parameterized reasoning is per definition inapplicable. The aggregate method still allows, however, the verification of any finite family with a single verification run. It does so by reporting the parameter values for which the property fails. The cost of the single verification run depends on how much the systems in the family diverge: the more the members of the family have in common, the closer is the verification cost to the cost of verifying only the largest of the systems, rather than of all systems.

1.3 Roadmap

This dissertation has four parts.

Part I continues with background information about model checking and abstraction. Chapter 4 is specifically devoted to symmetry reduction, as it is essential for understanding the material presented in this dissertation.

Part II describes new and improved principal approaches to symmetry reduction, suitable for symbolically represented systems. They are "principal" in the sense that they assume perfect symmetry, as defined in chapter 4. Chapter 5 presents *dynamic symmetry reduction*, a technique that avoids the orbit problem by avoiding the construction of an abstract model. Approaches based on *counter abstraction* are discussed in chapters 6 and 7. In chapter 8 we take a break from the theory and present the DySyRe tool that implements many techniques developed in this part of the dissertation. Chapter 9 provides experimental evidence for the effectiveness and efficiency of the methods presented in part II, both for each method alone and in comparison among them.

Part III describes methods that extend the scope of symmetry reduction. Chapter 10 generalizes principal techniques to allow for minor glitches in the symmetry of the system. This chapter focuses on explicit-state, not symbolic, model checking under symmetry. Chapter 11 discusses parameterized verification and presents an approach towards solving a feasible and practically relevant instance of the parameterized verification problem: verification with a bounded-size parameter.

Part IV concludes with a summary of the results of this dissertation, and a discussion of open problems (strongly related to the topics of this dissertation) and future research (loosely related to the topics of this dissertation).

With very few exceptions, proofs to theorems are included, either in the main text along with the theorem, or—if bulky—in appendix A. The rest of the appendix contains mainly code examples. The purpose of appendix B is to demonstrate how to model systems in the DySyRe symbolic model checker (introduced in chapter 8). To this end, the chapter presents the description of a variant of the Readers-Writers synchronization problem. In appendix C we list the code for a queuing lock example by J. Mellor-Crummey and M. Scott, which is used in various parts of this dissertation for demonstration purposes.

A dissertation that claims to substantially improve previous results on a particular problem must examine prior work in detail. We discuss such work at the end of the individual chapters in parts II and III, in a separate section with bibliographic notes, in order to be able to compare the details with our own techniques. In the main text we use citations sparingly; full credit is given in those reference sections.

Chapter 2

Model Checking

Overview. In this chapter we provide background on model checking and on symbolic system representation. We introduce Kripke structures, binary decision diagrams, temporal logics and model checking algorithms. The reader familiar with these concepts is welcome to skip this chapter; throughout this dissertation we use standard notation to denote standard concepts.

Model checking generally refers to the act of determining the truth value of a formula in a given environment that defines the free variables in the formula. Model checking can therefore be seen as an implementation of the semantics of the logic in which the formula is given. More specifically, the term model checking was coined in the context of *temporal logics* (section 2.2), which are interpreted over finite-state transition graphs called *Kripke structures* (section 2.1). The combination of temporal logics and Kripke structures is a crucial scenario in practice, for the following three reasons: (i) many interesting programs can be modeled using Kripke structures (perhaps after simplification), (ii) many interesting properties of programs can be expressed in propositional temporal logics, and (iii) the model checking problem for this constellation is decidable. In this dissertation, the term model checking is used to mean propositional temporal logic model checking over finite-state Kripke structures.

Thanks to decidability, there is an algorithm that takes a temporal logic formula and a Kripke structure and outputs "yes"—the formula evaluates to *true* over the structure—or "no"—it doesn't. If it does, we usually simply say the structure *satisfies* the formula. In light of this terminology, it is important to distinguish model checking from satisfiability checking, where the input is a formula, and the goal is to check whether a satisfying structure exists. The worst-case complexity of satisfiability checking for common temporal logics is as least a high as that of model checking.

Model checking is—in principle—complete, i.e. a failed proof attempt means the formula is indeed violated. We would like the model checker to produce evidence of the violation in such cases. This evidence is called a *counter example.* The possibility of generating it is one great asset of model checking, as it facilitates debugging of designs immensely.[1] For the common case of invariance properties—something that is to hold at any time during execution—, a counter example takes the form of a path to a system state exhibiting the violation. We note that a counter example is not possible or feasible for all formulas. For instance, a counter example for a formula that claims the reachability of a state with some property is tantamount to a *witness* for any such state's unreachability, which cannot in general be delivered succinctly.

Historical note. In mathematical logic, there are (unfortunately) many ways of saying that a formula evaluates to *true* in some environment. One is to say that the environment is a model of the formula. This formulation originally led to the term model checking. In verification, this use of the word *model* can be confusing, since

[1] In fact, although originally designed as a proof technique, model checking has become really popular as a bug-finding method. Model checking runs on immature designs that end with "no bug found" are often grounds for suspicion.

model also refers to a representation of something, in the most general sense. For example, a transition graph can be a model of a program. This sense of *model* has nothing to do with satisfaction and is in fact meaningful beyond model checking. To avoid the potential for ambiguity, in this dissertation we use *model* always in the representation sense (section 2.1) and say a structures *satisfies* a formula if it does.

2.1 Finite-State Models of Systems

Model checking can be informally described as a sophisticated exhaustive graph search method. Exhaustiveness can generally only be guaranteed for finite objects. Thus, to apply model checking we must represent the program or system in question as a finite graph-like object. Such objects, called *Kripke structures*, fulfill two purposes: (i) they describe the behavior of the system through transitions between states, and (ii) they ascribe a meaning to states by assigning to them basic *atomic propositions*. Purpose (ii) is what distinguishes Kripke structures from graphs: the atomic propositions provide the glue between the transition system and the temporal logic that is supposed to express properties over the transition system. We see how this is done in section 2.2.

2.1.1 Kripke Structures

Let AP be a finite set, interpreted as the universe of atomic propositions that a state can possibly satisfy. A Kripke structure over AP, often denoted by M (for "model"), is a triple (S, R, L), with the following components:

- S is a finite set,
- R is a subset of $S \times S$, and
- L is a function that maps each element of S to a subset of AP.

These components are interpreted as follows.

S is called the set of *states* of the Kripke structure M. It more or less precisely reflects the set of configurations that the system under investigation can be in. In practice, in order to be able to apply model checking we must be able to describe a run of the system as a sequence of state changes. Thus, the first step in modeling is to assign a notion of state to the system. Suppose, for example, the system is given as a program over a set of finite-range variables that are changed through assignments. One can define a state of this system as one particular valuation of all program variables. We emphasize that model checking may well be applicable to a system with apparently infinitely many configurations; the art of model checking is to map these configurations into a finite set of states that retain enough system information to allow the verification of the property in question.

R is called the *transition relation* of the Kripke structure M. Since S is finite, R is also finite. Set R reflects the behavior of the system under investigation. Once the set of states S is defined, the definition of a transition is usually automatic: every atomic behavior that causes a state change is a transition. A nonempty sequence of states $p = (p_0, p_1, \ldots)$ is called a *path* provided that adjacent states along p form a transition, i.e. for every $i \geq 0$, $(p_i, p_{i+1}) \in R$. Paths may be finite, in which case i's range is restricted in the obvious way. The *length* of a finite path p, denoted $|p|$, is the number of its transitions; a path of a single state thus has length zero. We denote by $p_{i\rightarrow}$ the *suffix* of p starting at position i; for a finite path p, this is defined only for $i \leq |p|$.

Since R is a relation, as opposed to a transition function, we can represent *reactive* systems: given a specific system configuration, the step to take may depend on an environment choice that the system has no control over. At the Kripke structure level, this means that the state that models the configuration has several possible successors—the structure is *nondeterministic*. On the other hand, a configuration may have no successor, indicating rightful termination of the system or a deadlock; the corresponding Kripke structure fails to be *total.* Curiously, a non-total structure may satisfy temporal properties in an unintuitive way; we discuss this phenomenon in section 2.2.4.

Kripke structures also allow us to model ongoing behavior. The goal of operating systems and flight controllers is not, as in traditional sequential programming, to compute a value, print it and quit, but rather to monitor and regulate the interplay of system components such as processes. Such systems are designed to run forever; termination may signal a crash or a deadlock. At the Kripke structure level, ongoing computation is modeled by infinite paths, i.e. by paths with cycles.

L is called the *labeling function* of the Kripke structure M. It is mainly there for formal reasons, to facilitate the definition of the semantics of temporal logics with respect to Kripke structures. Such logics express properties that change over time, along paths in the structure. The base case of defining their semantics is what is true at the current state. This information is in practice extracted directly from the state, i.e. from the values of the state variables. The labeling function abstracts this process by assigning to each state a subset of the set AP of predefined atomic propositions, namely those propositions assumed to be true at that state.

Suppose we are designing an operating system and are contemplating whether we need a disambiguation mechanism for simultaneous resource requests. We thus want to check whether states are reachable in the system with the property that two processes have a pending resource request. This property, call it *contention*, is atomic in the sense that to evaluate it of a state, we need to look at that state only, not at its successors or predecessors. The labeling function formally assigns the predicate *contention* to all states satisfying this property.

A Kripke structure sometimes has a fourth component: an initial state of the system. Temporal logic formulas that require a unique "start state" (most notably

CTL, see section 2.2.2) can then by default use the initial system state as the start state. In this dissertation we omit the initial state from the definition of the Kripke structure. The reasons are that the convention to use the initial state as start state fails to make sense if a system has several initial states, and that some logics do not require a start state at all (most notably LTL, see section 2.2.1).

Figure 2.1 shows an example Kripke structure of five states and six transitions. States are labeled with the atomic propositions true at them; one state has an empty label. As we see, different states can be labeled with the same set of atomic propositions, and a single state can be labeled with several atomic propositions. The state labeled $\{C, D\}$ has two successors—the structure is nondeterministic. Each state has at least one successor—the structure is total. The dashed edges form a cycle and can be unfolded into an infinite path.

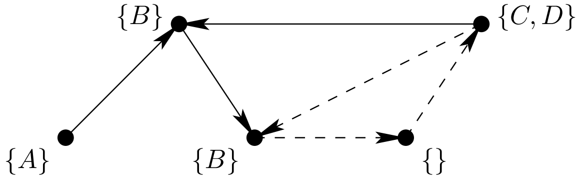


Figure 2.1: A Kripke structure

Kripke structures form the theoretical foundation for modeling systems as a finite-state machine. In the rest of section 2.1 we review how such structures are represented in a computer program such as a model checker. The motivation behind many of these design decisions is the state-explosion problem.

2.1.2 Explicit-State Representations of Kripke Structures

As a Kripke structure is a graph-like object, it stands to reason to use well-studied graph data structures to encode a Kripke structure. Early implementations of model checking translated the given program into an adjacency list representing the transition relation and a “dictionary” to look up atomic propositions true at a state. This approach has an obvious disadvantage: it is exposed mercilessly to state explosion in that every state that is just conceivable amounts to a piece of memory being occupied. Any valuation of the program variables in accordance with the variables' declarations constitutes a conceivable state. If the conceivable state space is huge, such an implementation becomes infeasible.

The main reason why it is not necessary to pre-generate the conceivable state space is that, given some initial state, there is no need to consider unreachable states. In practice, the unreachable part of the state space can be significant. Consider a system of n processes, each of which can be in l different local states. Suppose we have l counter variables $n_1, \ldots, n_l \in [0..n]$, such that n_L counts how many processes are currently in local state L. The conceivable state space of the counter system is of size $(n+1)^l$. By construction, however, the counters satisfy the constraint $\Sigma_{L=1}^{l} n_L = n$. There are much fewer than $(n+1)^l$ counter tuples satisfying this constraint. Thus, the reachable state space is much smaller than the conceivable one.

In practice, model checking is therefore rarely done by first computing and representing a set of states and of transitions in a graph data structure such as an adjacency list. A program can itself be viewed as a high-level and compact representation of the Kripke structure's transition graph. Given a Kripke structure state, we can compute successors by mapping the state back to a program state, applying the rules of the program to obtain successor program states, and mapping those to Kripke structure states. This may seem costly, but if the correspondence between program states and Kripke structure states is tight, the mappings may amount to the identity function. This procedure is known as *on-the-fly model checking.* Out of the three components (S, R, L) of a Kripke structure, none is realized in full:

States are only created on the fly, as they are encountered, and R and L exist only conceptually. In particular, instead of looking up whether a state is labeled with P, we apply the expression defining P to the state in question. For complex properties that require exploring parts of the state space several times, we thus may evaluate this expression several times for the same state. This evaluation is often linear in the size of the state and is thus feasible.

A Kripke structure encoding in which each state is represented explicitly by its own piece of memory is called an *explicit-state* encoding. Surprising at first, it is possible to encode a set of m states in much less memory than, say, a list of length m would require. Such implicit representations are known as symbolic and are discussed next.

2.1.3 Symbolic Representations of Kripke Structures

The main idea to achieve a Kripke structure representation more compact than an explicit enumeration of states is to use constraints. Such a representation has become known as *symbolic* since it is based on formulas. Consider a system with two variables, $v, w \in \{A, B, C\}$. The set of states $Z = \{(v, w) : v = w\}$, where the variables have the same value, can be represented extensionally by enumerating the three states it contains: $\{(A, A), (B, B), (C, C)\}$. It can also, however, be represented using its defining constraint: $v = w$. This method is sound and complete: every closed-form Boolean constraint over the program variables represents a unique set of states. Conversely, every set Z of states can be represented by a constraint, for example by the constraint $\bigvee_{s \in Z} \mathit{constr}(s)$, where $\mathit{constr}(s)$ encodes the state s.

This idea can be extended to represent sets of transitions using constraints. A transition $(s, t) \in R$ is written as the constraint $\mathit{constr}(s) \wedge \mathit{constr}(t)$; to be able to distinguish variables that define s from those defining t we attach a prime $'$ to each variable used in defining t. For example, consider a system of two Boolean variables x and y and the statement **if** x **then** $y := \mathit{true}$. Encoding s using x and y, and t using x' and y', the statement defines the set of transitions (s, t) that obey the relation

$$R(s,t) \quad = \quad R(x,y,x',y') \quad = \quad ((x \wedge y') \vee (\neg x \wedge y' = y)) \;\wedge\; x' = x\,. \tag{2.1}$$

Variables x and y are sometimes called *current-state* copies of the state variables, x' and y' then are the *next-state* copies. The example shows an important characteristic of symbolically represented transitions: variables that do not change must be constrained such that their current-state copy equals their next-state copy. For example, variable x is invariant under the **if** statement above. In many programming languages, it is implicit that unassigned variables are unchanged. In contrast, omitting the constraint $x' = x$ in the expression above results in a transition where the next-state value of x is nondeterministically chosen—it is unconstrained.

Once the transition relation is represented as a Boolean formula over two copies of the state variables—current and next—, we have to think about a suitable data structure to encode this formula. Since we want to use the transition relation for model checking, the choice of data structure depends on the operations we want to perform on such a formula. In addition to basic set-theoretic operations such as union and intersection, we need to be able to test two formulas for equivalence, and we need to be able to compute successors and predecessors of states. In principle, a straight-forward conjunctive normal form (CNF) suffices; there are model checking algorithms that use this encoding. However, equivalence checking can be expensive in CNF. We discuss next an alternative encoding that is very popular with model checking algorithms and also used extensively in this dissertation.

2.1.4 Binary Decision Diagrams

Binary decision diagrams—BDDs—were introduced by R. Bryant [Bry86] as a flexible notation for Boolean formulas that allows all of the above operations, in particular equivalence checking, fairly efficiently. In addition, it turned out that for many practical systems, the corresponding transition relation can be represented succinctly in such a diagram. “Succinctly” here means: with a number of diagram entries that is far below the number of truth value assignments that make the formula true, i.e. far below the number of objects represented by the diagram. If $\mathbf{P} \neq \mathbf{NP}$, we cannot expect succinctness for *all* propositional formulas; crucial is succinctness in many practical cases.

Representing a Formula as a BDD

Given a formula, its BDD is obtained from an intermediate representation called *binary decision tree* through a sequence of simplifications that remove redundancy in the tree. Suppose k variables occur in the formula. The decision tree is a complete binary tree of depth k. All nodes at level i are labeled with the ith variable. The left subtree of a node corresponds to the subformula obtained by instantiating the variable attached to the node by *false* (0) in the formula, analogously for the right subtree. This way, for any node at level $k + 1$, each variable in the formula has a value. We can label the node with F if the formula evaluates to *false*, and T otherwise. Figure 2.2 (left) shows the binary decision tree for formula (2.1).

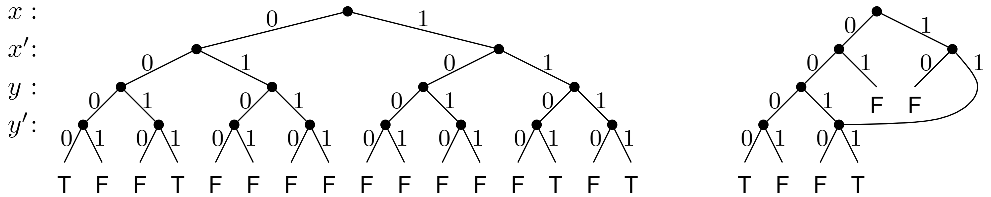


Figure 2.2: Binary decision tree (left) and decision diagram (right) for formula (2.1)

Such a binary tree is of size roughly 2^{k+1} and thus not very compact. It contains, however, much redundancy. For example, consider the node corresponding to the partial assignment $(x, x') := (0, 1)$ in the left half of the tree in figure 2.2. All leaves reachable from this node are labeled F, so we can as well label the inner node F and stop expanding the tree. Now consider the node corresponding to the partial assignment $(x, x', y) := (1, 1, 1)$ on the far right of the tree. Its subtree is not redundant, but identical to the subtree of the node corresponding to the partial assignment $(x, x', y) := (1, 1, 0)$. Thus we can ignore variable y and make the node corresponding to the partial assignment $(x, x') := (1, 1)$ point directly to (a single copy of) that subtree.

This procedure is performed repeatedly until no more simplification applies, resulting in the binary decision *diagram* on the right in figure 2.2. A binary decision diagram is a directed acyclic graph (DAG) that is generally much smaller than the original decision tree. For example, formula (2.1) is seen to be unsatisfiable if $x \neq x'$. This is reflected in the subtrees for $(x, x') := (0, 1)$ and $(x, x') := (1, 0)$ pointing to F; variables y and y' are not considered at all in those subtrees. In practice, the compression is taken a step further by having only a single leaf labeled F and a single leaf labeled T (not done in figure 2.2 for legibility).

BDD Variable Order

The succinctness of the BDD in figure 2.2 owes, of course, to the choice to consider variables x and x' *before* y and y'. Indeed, if we changed the variable order to y, y', x, x' and constructed the corresponding BDD, we would see that it is by two nodes larger. The dependence of BDDs on a favorable variable order is one of the disadvantages of this data structure, in particular since it is generally expensive to determine what a good order is. In practice, implementations use heuristics such as keeping current-state and next-state copies of a variable close together in the order. Another option is *dynamic reordering*: if, as a result of Boolean operations, a BDD grows too large during its lifetime, it is converted into an equivalent one with a different variable order, hoping that the new order allows for a more compact diagram.

It has proved useful to require that in all BDDs existing in the program, variables are read in the same order from the root to any leaf, up to omissions of variables.[2] One motivation for this requirement is that then operations that combine BDDs can be performed reasonably efficiently. Another motivation is that given a fixed variable order, any truth table has a unique BDD. This has the consequence that two formulas are equivalent exactly if their BDDs are identical. This property greatly simplifies equivalence checking, an important operation in model checking algorithms. In practice, BDD packages may even attempt to never keep two copies of the same BDD. That is, if some operation results in a BDD that already exists, the return value of the operation is simply a pointer to the existing BDD. It requires some runtime commitment to stick to this protocol. The benefit is not only minimum memory needs, but also equivalence checking now being a constant-time operation: the pointers of the two BDDs are compared.

Operations on BDDs

We finally sketch how operations important in connection with Kripke structures can be implemented on BDDs. We have already discussed checking equivalence of two formulas. A formula is satisfiable exactly if its BDD is different from the BDD for the formula *false*, which is just an isolated leaf node labeled F. The negation of a formula is computed by re-labeling the F-node T and vice versa. Two-place Boolean functions can be implemented with the help of the *Shannon expansion* with respect

[2] Due to the fixed variable order, BDDs are also called *ordered* binary decision diagrams.

to any variable x_i:

$$\begin{array}{rcl} f(x_1, \ldots, x_k) & = & (\neg x_i \;\; \wedge \;\; f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_k)) \\ & \vee & (\;\; x_i \;\; \wedge \;\; f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_k)) \end{array} \tag{2.2}$$

This equivalence suggests a recursive procedure to compute, say, the conjunction $f_1 \wedge f_2$ of two Boolean formulas given as BDDs. We rewrite both f_1 and f_2 using Shannon expansion with x_i being the variable at the root level. Since $\wedge$ distributes over the $\vee$ in equation (2.2), we have split the problem $f_1 \wedge f_2$ into two subproblems, amounting to the conjunction of the two respective parts of the Shannon expansion. When the recursion reaches the leaves, the result is the BDD for the constant function *false* or *true*, depending on the label of the leaf.

Suppose Z is a set of states given symbolically as a Boolean constraint, which in turn is represented as a BDD. For model checking, it is critical to be able to compute the set of successors, with respect to a transition relation R, of states in Z. This is done in three steps: (i) Compute the set $A = R \wedge Z$ of pairs (x, x') in R such that $x \in Z$. We now have to project the pairs in A to their second components. To this end: (ii) Compute $B = \exists \vec{x} : A$, which eliminates, by existential quantification, the current-state variables $\vec{x}$ in pairs in A. B is the result we seek, except that it is expressed in terms of next-state variables (second components of R). Thus: (iii) Rename every (next-state) variable in B to its current-state counter part. Predecessor computations and a few other operations typically needed in connection with Kripke structures can be performed on BDDs in a similar fashion.

2.1.5 Modeling Systems with Many Components

This dissertation considers systems of many concurrently existing components, such as processes in an operating system. Such a system can be described by specifying the behavior of each component, along with a characterization of the concurrent execution model. Each component can be viewed as an *open* subsystem, one whose behavior is determined in part by an environment, i.e. by the other components. The overall concurrent system, on the other hand, is closed.

In this dissertation we always assume an asynchronous concurrent execution model. This means that a state change in the system (and, equivalently, in the derived Kripke structure) is given by a change in exactly one component. The reason for this assumption is that this execution model is most common for concurrent systems of processes, which are the main area of application of the results of this dissertation. Synchronous execution is important with digital circuits, where some or all gates may fire at the same time.

Each system component usually has a set of variables that it manipulates; we call them *local* variables. Each valuation of the local variables determines the *local state* of the component. For synchronization purposes, the system has in addition a set of *global* variables. Formally, assuming l local states, the state space of such a system's Kripke structure is given by a vector $\vec{v}$ of global variables, say with combined domain V, followed by a vector of n variables specifying the local state of each of the n components: $S = V \times [1..l]^n$. We can write a state s of this Kripke structure in the form $(\vec{v}, s_1, \ldots, s_n)$, where $s_i \in [1..l]$ is the local state of component i. For a local state L, we often use the notation L_i to denote the atomic formula $s_i = L$. For example, the expression $\forall i : L_i$ expresses that every component of the system resides in local state L.

In this dissertation, we in particular consider systems of many *replicated* components, where the behavioral descriptions of the components are essentially the same. The description is in this case a program that is parameterized by the name of the executing component. Allowing such a parameter increases the expressiveness of this type of system model. For example, we can model a solution to the mutual exclusion problem using a global token variable that contains the name of the next process allowed to enter the critical section. We can enforce fairness by incrementing the token in a circular fashion every time a process leaves the critical section.

Using the abstraction mechanism from above, which suggests to compress the valuations of local variables into local states, we can describe the parameterized program as a graph known as a *synchronization skeleton* [CE81]. We show an example in figure 2.3. Each node in the skeleton represents a local state, each edge a change between local states. To achieve synchronization, a skeleton's edges can be labeled with guards (shown in the figure above the edge) and actions (shown below the edge). Guards are Boolean-valued expressions on local states of processes and global variables. Actions are assignments to global variables. The actions are executed after the local state change. The skeleton in the figure allows process i to enter its critical section C if the token currently points to the process ($tok = i$). Upon leaving C, the token is passed on to the next process.

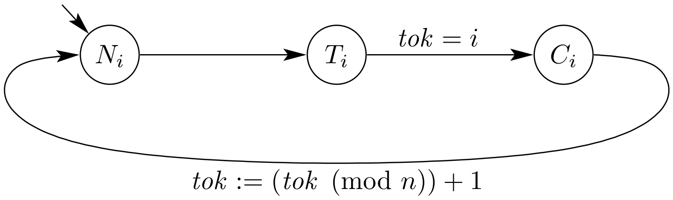


Figure 2.3: Synchronization skeleton for a solution to the Mutual Exclusion problem

We can think of a synchronization skeleton as a succinct notation for a concurrent program where valuations of local variables of a process are abstracted into a local state, and assignments to those variables are represented as local state changes. Sequential code executed by a process atomically (not interleaved with other processes) is abstracted into a single local transition. Given a number n, a synchronization skeleton gives rise to a Kripke structure modeling a system of n asynchronously executing processes. We make use of the skeleton notation in chapter 6, at which time we give a formal definition of the derived Kripke structure.

This concludes the discussion of finite-state models of systems and how they are represented in programs suitable for model checking. The second ingredient of any formal verification procedure is a specification of the properties that we want to check the system for. We treat this topic next.

2.2 Specifying Properties of Systems

In the previous section we have seen how a system can be modeled as a Kripke structure, which is a state transition graph augmented by atomic properties that are attached to the states. An (infinite) computation of the system is thus represented as an (infinite) path through the transition graph. Every state along the path is labeled with some atomic propositions. The path can be viewed as a timeline that characterizes each point in time through the atomic propositions true at that time.

A property of a computation specifies changes in the truth of atomic propositions. One aspect of change is change over time, addressed using *temporal operators.* They allow us to express that something is true *next time* (after one transition), or *at some time* (after a finite number of transitions), or *always* (now and after every transition). A different aspect is change of truth due to the branching nature of nondeterministic programs. For example, consider a concurrent system of two processes. The truth of the property, "After one time unit, process 1 performs I/O", likely depends on the direction in which the system goes, i.e. which process is scheduled for execution. More generally, branching causes the existence of infinitely many futures, i.e. paths starting from the current state. We may want to express temporal properties for specific futures, which is done using *path quantifiers.* At the coarsest level, we can state that there exists a future satisfying some temporal property, or that all futures do.

Temporal logics are defined by selecting which temporal operators and path quantifiers are allowed in what combinations. We formally describe two of the more frequently used such logics, known as LTL and CTL.

2.2.1 Linear Temporal Logic

Linear temporal logic (LTL) was the first to be used in describing and reasoning about reactive programs. It appeals through its simplicity, as it allows temporal operators, but no path quantifiers:

Definition 1 *Given a set of atomic propositions AP,* LTL *is the smallest set of formulas satisfying the following conditions:*

(base formulas) *The propositional constants false and true are LTL formulas. For $P \in AP$, P is an LTL formula.*

(closure under propositional connectives) *If f is an LTL formula, so is $\neg f$. If g and h are LTL formulas, so are $g \wedge h$, $g \vee h$, etc.*

(closure under temporal operators) *If f is an LTL formula, so are $\mathsf{X}\, f$ and $\mathsf{G}\, f$. If g and h are LTL formulas, so is $g \,\mathsf{U}\, h$.*

An LTL formula is evaluated over an infinite path through a Kripke structure:

Definition 2 *Given a set of atomic propositions AP, let f be an LTL formula, $M = (S, R, L)$ a Kripke structure over AP and p an infinite path of M. Path p is said to* satisfy *f, written $p \models f$, depending on the form of f as follows:*

1. *$p \models true$. For $P \in AP$, $p \models P$ iff $P \in L(p_0)$.*

2. *$p \models g \wedge h$ iff $p \models g$ and $p \models h$, analogously for the other connectives.*

3. *$p \models \mathsf{X}\, h$ iff $p_{1\rightarrow} \models h$.*

4. *$p \models \mathsf{G}\, h$ iff for all i, $p_{i\rightarrow} \models h$.*

5. *$p \models g \,\mathsf{U}\, h$ iff there exists i such that $p_{i\rightarrow} \models h$ and for all $j < i$ it is $p_{j\rightarrow} \models g$.*

The case $f = \mathit{false}$ is not mentioned; thus no path satisfies the formula *false*. An atomic proposition P is satisfied if the path's first state is labeled with P. The semantics of propositional connectives is standard. X is the *next-time* operator; $p_{1\rightarrow}$ denotes the suffix of p after the first state. The formula $g \,\mathsf{U}\, h$ can be read as "g until h". It expresses that there is a moment i along the path at which h holds, and at all moments before that, g holds. Additional temporal operators can be introduced; a common one is $\mathsf{F}\, h := \mathit{true}\, \mathsf{U}\, h$. G and F can intuitively be interpreted as *always* and *eventually*.[3] We note that for finite paths some adjustments are necessary; for example $p \models \mathsf{X}\, h$ requires $|p| \geq 1$.

Below are some typical examples of LTL formulas; we assume suitable atomic propositions exec_s, *req*, *grant* and "$x < 0$":

(A). "Statement s is infinitely often executed." $\mathsf{GF}\, \mathit{exec}_s$

(B). "Every resource request is followed by a grant." $\mathsf{G}(\mathit{req} \Rightarrow \mathsf{F}\, \mathit{grant})$

(C). "There is a point after which never $x < 0$." $\mathsf{FG}\, \neg(x < 0)$

Example (B) is a good occasion to make oneself aware of the distinction between the Boolean connective $\Rightarrow$ and the *next-time* temporal operator X.

It is important to keep in mind that the semantics of LTL is defined with respect to a single computation path. Reactive programs usually have infinitely many such paths. When given a program, modeled as a nondeterministic Kripke structure, and an LTL formula, it is up to the implementation of the model checker to decide whether "the formula is true" means "it is true for some computation" or "it is true for all computations", or maybe even something else. Usually, though, it is intended to mean that the formula holds along all paths; we sometimes say an LTL formula has an implicit universal path quantifier.

[3] As a mnemonic, we can think of G and F as *globally* and *finally*.

The convention of the implicit universal path quantifier causes a dilemma: when evaluated over a Kripke structure, LTL is not closed under negation. For example, while the property "P is invariantly true" can be written as $\mathsf{G}\,P$ (with an implicit universal path quantifier added by the model checker), the negation of this property, "$\neg P$ is reachable along some path", is not expressible in LTL, again with the implicit universal path semantics. To use an LTL model checker to check this formula, we need to check the original formula $\mathsf{G}\,P$ and negate the outcome "manually".

Obviously, this manual negation generally works only as long as the formula we intend to check over the structure does not use both universal and existential path quantifiers. This limitation of LTL was one of the motivating factors for introducing another temporal logic, CTL.

2.2.2 Computation Tree Logic

Computation tree logic (CTL) followed LTL a few years later in an attempt to allow the explicit specification of branching in a program. It turned out that if the way temporal operators and path quantifiers can be combined is restricted to certain forms, we obtain a logic for which the complexity of model checking is actually much lower than it is for LTL.

Definition 3 *Given a set of atomic propositions AP,* CTL *is the smallest set of formulas satisfying the following conditions:*

(base formulas) *The propositional constants false and true are CTL formulas. For $P \in AP$, P is a CTL formula.*

(closure under propositional connectives) *If f is a CTL formula, so is $\neg f$. If g and h are CTL formulas, so are $g \wedge h$, $g \vee h$, etc.*

(closure under modalities) *If f is a CTL formula, so are $\mathsf{EX}\,f$ and $\mathsf{EG}\,f$. If g and h are CTL formulas, so is $\mathsf{E}(g\,\mathsf{U}\,h)$.*

A CTL formula is evaluated over an infinite computation tree of a Kripke structure:

Definition 4 *Given a set of atomic propositions AP, let f be a CTL formula, $M = (S, R, L)$ a Kripke structure over AP and $s \in S$. Structure M is said to* satisfy *f with respect to s, written $M, s \models f$, depending on the form of f as follows:*

1. *$M, s \models true$. For $P \in AP$, $M, s \models P$ iff $P \in L(s)$.*

2. *$M, s \models g \wedge h$ iff $M, s \models g$ and $M, s \models h$, analogously for the other connectives.*

3. *$M, s \models \mathsf{EX}\,h$ iff there exists $t \in S$ such that $(s, t) \in R$ and $M, t \models h$.*

4. *$M, s \models \mathsf{EG}\,h$ iff there exists a path p of M such that $p_0 = s$ and for all i, $M, p_i \models h$.*

5. *$M, s \models \mathsf{E}(g\,\mathsf{U}\,h)$ iff there exists a path p of M and an index i such that $p_0 = s$, $M, p_i \models h$, and for all $j < i$, it is $M, p_j \models g$.*

Analogously to LTL, we can introduce EF as a special case of EU, namely $\mathsf{EF}\,h := \mathsf{E}(\mathit{true}\,\mathsf{U}\,h)$. We see that EX, EG, EU and EF essentially mean the same as LTL's X, G, U and F, respectively, except that no path is given; instead its existence is claimed. Therefore, EX, EG, EF and EU are called *existential modalities*; using negation we introduce the equally important *universal modalities*: $\mathsf{AX}\,h$ short for $\neg\,\mathsf{EX}\,\neg h$, $\mathsf{AG}\,h$ for $\neg\,\mathsf{EF}\,\neg h$, $\mathsf{AF}\,h$ for $\neg\,\mathsf{EG}\,\neg h$ and $\mathsf{A}(g\,\mathsf{U}\,h)$ for $\neg\,\mathsf{E}(\neg h\,\mathsf{U}(\neg g \wedge \neg h)) \wedge \neg\,\mathsf{EG}\,\neg h$.

We see that while CTL allows both temporal operators and path quantifiers, it does so only in a very disciplined way: temporal operators may not nest, they must individually be preceded by a path quantifier. The logic CTL* mentioned below allows arbitrary nesting. The motivation for extracting CTL as a sublogic of CTL* has to do with the complexity of model checking and is explained in section 2.3.2.

Below are some basic examples of CTL formulas:

(D). “Along some future, property P is true at some time.” $\mathsf{EF}\,P$

(E). “Along all futures, property P is true at some time.” $\mathsf{AF}\,P$

(F). “Along all futures, property P is always true.” $\mathsf{AG}\,P$

These three property schemata are very common in practical verification: (D) expresses *reachability* of (a state satisfying) P, (E) *inevitability* of P, and (F) *invariance* of P. The set of R-successors of a state z (or of a set of states Z) is known as the *image* of z (or of Z). Likewise, the set of R-predecessors is called *preimage*. Since $\mathsf{EX}\,f$ represents the set of predecessors of f-states, EX is called the *existential preimage* operator. Similarly, $\mathsf{AX}\,f$ represents the set of states s such that all successors of s satisfy f; AX is called the *universal preimage* operator.

2.2.3 CTL* and the Propositional μ-calculus

The logic CTL* combines the features of LTL and CTL to a formalism that is in fact more expressive than the union of the two. Roughly speaking, CTL* formulas allow arbitrary combinations of temporal operators and path quantifiers, except that temporal operators may not appear at the top level of the formula. For example, the formula $(\mathsf{AGF}\,P) \vee (\mathsf{EF}\,Q)$ satisfies this constraint and thus belongs to CTL*, but not to LTL or CTL. Moreover, it is not even equivalent to any LTL or any CTL formula. Despite this expressiveness, the complexity of CTL* model checking equals that of LTL model checking, although satisfiability checking is more expensive for CTL* than it is for the other logics.

A yet more expressive formalism is given by the (propositional) μ-calculus. At first glance, this logic looks different from the ones we have considered so far, as it does not use temporal operators or general path quantifiers. Instead, what gave it its name is the use of the operators μ and ν for the least and greatest fixpoints of a predicate transformer. Such a transformer is a mapping $\tau\colon 2^S \to 2^S$, i.e. it transforms a set of states (a predicate) to another set of states. If the expression $\tau(Z)$ is syntactically monotone, i.e. Z occurs under an even number of negations, then there exists a set Z^* such that $\tau(Z^*) = Z^*$. Such a set is called a *fixpoint* of τ. Moreover, there exists a fixpoint of τ that is contained in any other fixpoint of τ, called *least* and written $\mu Z.\tau(Z)$. Analogously, there exists a fixpoint of τ that contains any other fixpoint of τ, called *greatest* and written $\nu Z.\tau(Z)$. Such fixpoint operators are very powerful and in particular sufficient to encode the temporal aspects of LTL and CTL (for CTL, see section 2.3.2).

Model checking for the μ-calculus is an active area of research; its precise complexity is unknown. Currently existing (deterministic, non-randomized) algorithms are exponential in the size of the Kripke structure. Recent research has gradually brought down the exponent from, originally, the number of nested fixpoint expressions, to the fixpoint alternation depth (number of consecutive sequences of fixpoints of the same type), to one half of the alternation depth. There are conjectures that the μ-calculus model checking problem is intrinsically super-polynomial in the size of the structure. We return to the μ-calculus when we present an algorithm for CTL model checking in section 2.3.2.

2.2.4 Concluding Remarks

Temporal logics are a special type of *modal logics* (note: modal vs. model), which generally express the phenomenon of truth values changing as a function of certain modalities of life, such as time or space. Such logics were originally introduced by philosophers investigating non-absolute truth. They have become an active research field in computer science since A. Pnueli, in a milestone paper, showed how they can be used to describe and reason about program behavior [Pnu77]. An extensive survey of temporal and modal logics by A. Emerson can be found in [Eme90].

In this dissertation we mostly work with the standard versions of LTL or CTL. These versions can, however, be enhanced along various dimensions. A fairly straightforward addition is given by *past-time* operators, which are convenient to traverse a Kripke structure in forward direction. This can be advantageous when the forward branching degree of the structure is much lower than the backward branching degree, such as when the structure is roughly deterministic. Other enhancements affect the non-temporal part of the syntax and semantics of the logics. *First-order* (in contrast to propositional) versions allow variables, predicates, functions etc. in addition to atomic propositions. In *continuous-time* temporal logic, the timeline is not assumed to be discrete, but a dense number range such as the rationals or even reals. The spectrum of these enhancements ranges from "syntactic sugar" (as with the past-time operators) to the destruction of decidability (as with too liberal first-order forms).

We mentioned in section 2.1.1 that it is desirable for a Kripke structure to be total, i.e. every state should have a successor. The reason is that temporal logics are designed to express properties of infinite computations. While it is mathematically legal to evaluate properties over finite paths, the results may be absurd. Consider the LTL formula $\mathsf{GF}\,P$. Intuitively, if something is "always eventually" true, it means it holds infinitely often.[4] If the current state satisfies P and happens to have no successor, this formula evaluates to true. Worse, consider the CTL formula $\mathsf{AX}\,Q$. If there is no successor, this formula is true no matter what Q. This phenomenon is one instance of *vacuous* satisfaction and can be ruled out by requiring a structure to be total.

[4] For this reason, $\mathsf{GF}\,P$ is sometimes written as $\mathsf{F}^{\infty}\,P$.

2.3 Model Checking—Algorithms and Implementation

In the previous sections we have introduced Kripke structures and temporal logics as the principal means of expressing models of programs and properties about them. The goal of this section is to sketch how they can be put together in algorithms that solve the model checking problem for these logics, which are interpreted over Kripke structures. There are numerous algorithms for model checking; the choice depends on the temporal logic targeted and on the data structure used to represent Kripke structures. In this section we choose two representative algorithms, one for LTL and one for CTL, which are actually used in tools and which also play a rôle in this dissertation.

2.3.1 Automata-Theoretic LTL Model Checking

One popular approach to the LTL model checking problem is using Büchi automata. Such automata are standard finite-state automata; what differs is that their notion of acceptance is defined with respect to infinite words. It turns out that such automata can be used to represent both a Kripke structure and a temporal logic formula; solving the model checking problem is then a matter of applying standard automata-theoretic and graph-theoretic techniques.

A finite-state *(Büchi) automaton* is a quintuple $(\Sigma, Q, \delta, Q_0, F)$, specifying an *alphabet* Σ, a finite set of *states* Q, a *transition relation* $\delta \subseteq Q \times \Sigma \times Q$, a set of *initial* states Q_0, and a set of designated *accepting* states F. A *run* of the automaton on an infinite word $w \in \Sigma^\omega$ is an infinite sequence of states $r = (r_0, r_1, \ldots)$ such that $r_0 \in Q_0$ and for each $i \geq 0$, $(r_i, w_i, r_{i+1}) \in \delta$. In words, a run on w is a path through the automaton that follows edges with labels given by w. A run is *accepting* if it has infinitely many occurrences of accepting states. Since Q is finite, an accepting run actually contains at least one accepting state infinitely often. An infinite word w is accepted by the automaton if there is an accepting run on w. The *language* of an automaton is the set of infinite words it accepts.

Such an automaton bears obvious resemblance with a Kripke structure: it is a form of state transition diagram, with infinite paths and nondeterminism. The difference between the two models is the labeling of edges with symbols vs. the labeling of states with atomic propositions. It is straightforward to turn a Kripke structure $M = (S, R, L)$ into an automaton A_M over the alphabet 2^{AP}. States and transitions are retained; an edge of the automaton is labeled with the atomic propositions that are true in the *successor* state.[5] We let every automaton state be accepting. As a result, the language of the automaton is the set of all infinite paths through the Kripke structure, projected to the atomic propositions true at states along the paths: $\mathcal{L}(A_M) = \{L(p) : \ p \text{ is a path in } M\}$.

An LTL formula f is represented as a Büchi automaton A_f (over the same alphabet 2^{AP}) such that $\mathcal{L}(A_f) = \{L(p) : p \models f\}$. That is, the property automaton accepts exactly the signatures of paths that satisfy f. The general algorithm for building such an automaton is known as *tableau construction*. This algorithm is quite involved; instead of presenting it here, we give examples for typical LTL formulas using the basic temporal operators (figure 2.4). The set of atomic propositions AP in these examples is $\{P\}$, $\{P\}$ and $\{P, Q\}$, respectively.

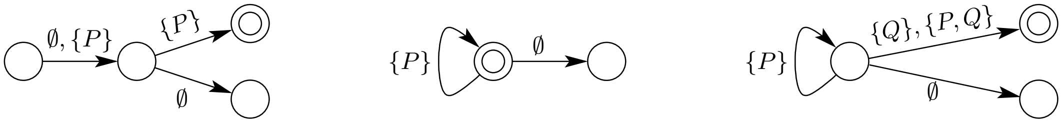


Figure 2.4: Büchi automata representing the formulas $\mathsf{X}\,P$, $\mathsf{G}\,P$ and $P\,\mathsf{U}\,Q$. A node without outgoing edges is meant to have a self-loop labeled with all subsets of AP

Given an LTL formula f, consider now a model checking problem of the form $M, s \models f$ with the "implicit universal path quantifier" semantics. The problem is equivalent to checking that no path through M, starting at s, satisfies $\neg f$.

[5] Some adjustments are necessary to correctly simulate the labeling of the initial state, i.e. the state with respect to which a formula is to be verified.

To this end, we translate M into an automaton A_M, assuming s as the initial state of M. We also translate $\neg f$ into an automaton $A_{\neg f}$ using the tableau construction. The goal is then to check that no run of A_M (which is by construction accepting) violates f, i.e. that is an accepting run of $A_{\neg f}$. We thus want to check whether $\mathcal{L}(A_M) \cap \mathcal{L}(A_{\neg f}) = \emptyset$.

We have now reduced LTL model checking to the sequential composition of two standard automata-theoretic problems: (i) the intersection problem: given two automata (here: A_M and $A_{\neg f}$), build a *composite* automaton that accepts the intersection of their languages, and (ii) the emptiness problem: given an automaton (here: $A_M \cap A_{\neg f}$), check whether its language is empty; if not generate a run that is accepted. Problem (i) is solved by building an automaton that executes A_M and $A_{\neg f}$ in lock-step. Problem (ii) is solved by realizing that an infinite path through one accepting state exists exactly if there exists a reachable accepting state whose strongly connected component is nontrivial. Thus, (ii) is solved by a reachability analysis on the SCC-quotient of $A_M \cap A_{\neg f}$.

Counter examples. Consider now an accepting run of the composite automaton $A_M \cap A_{\neg f}$. By construction, this run represents a path through M that satisfies $\neg f$, i.e. that violates f. Such a path thus serves as a counter example to the original model checking problem $M, s \models f$. The path can be written down as a finite path to the reachable accepting state with a nontrivial strongly connected component, followed by any loop through this component. Such a loop exists since the component is nontrivial.

Complexity. This algorithm for LTL model checking can be shown to have worst-case complexity $\mathcal{O}(|M| \cdot 2^{|f|})$. Converting a Kripke structure M into an automaton amounts to little more than writing down the result A_M. Converting an LTL formula f into the property automaton A_f, the tableau construction, is worst-case exponential in the size of the formula. Despite this complexity, the LTL model checking algorithm presented here is widely used. The final property automaton is often much smaller than the intermediate tableau constructed. Further, since $|f|$ is usually much smaller than $|M|$, the exponential complexity in $|f|$ has a limited impact in practice. Finally, the algorithm presented is suitable to be implemented space-efficiently in an on-the-fly fashion (see end of section 2.1.2).

2.3.2 Symbolic CTL Model Checking

In this section we sketch a model checking algorithm that exploits the special structure of CTL formulas. As mentioned in section 2.2.3, CTL can be embedded into the μ-calculus using fixpoints of predicate transformers, like all other temporal logics presented so far. Before we show the CTL embedding, we introduce a common (abuse of) notation. Consider a fixed Kripke structure $M = (S, R, L)$. When working with predicate transformers, we can extend CTL's preimage modalities EX and AX to operate on a set of states $Z \subseteq S$ (as opposed to on a formula) as follows:

$$\mathsf{EX}\,Z \;=\; \{s : \exists t :\; R(s,t) \;\wedge\; t \in Z\}\,,\ \text{and} \tag{2.3}$$

$$\mathsf{AX}\,Z \;=\; \{s : \forall t :\; R(s,t) \;\Rightarrow\; t \in Z\}\,. \tag{2.4}$$

Given a fixed Kripke structure M, a CTL formula can be identified with the set of states that satisfy it: $f \mapsto \{s : M, s \models f\}$. The formula notation is often more elegant than the set-theoretic one. In light of this elegance, we allow an expression like $f \vee \mathsf{EX}\,Z$, which is a mixture of logical and set-theoretic notation, to stand for the set of states $\{s : M, s \models f\} \cup \mathsf{EX}\,Z$.

What makes CTL's embedding into the μ-calculus special is the simplicity of the fixpoint expressions used to encode the logic's basic modalities:

$$\begin{array}{lcllcl} \mathsf{EF}\,h & = & \mu Z.h \vee \mathsf{EX}\,Z & \mathsf{AF}\,h & = & \mu Z.h \vee \mathsf{AX}\,Z \\ \mathsf{EG}\,h & = & \nu Z.h \wedge \mathsf{EX}\,Z & \mathsf{AG}\,h & = & \nu Z.h \wedge \mathsf{AX}\,Z \\ \mathsf{E}(g\,\mathsf{U}\,h) & = & \mu Z.h \vee (g \wedge \mathsf{EX}\,Z) & \mathsf{A}(g\,\mathsf{U}\,h) & = & \mu Z.h \vee (g \wedge \mathsf{AX}\,Z) \end{array} \tag{2.5}$$

(The period "." in the fixpoint operators has the lowest binding power.) For example, the equation $\mathsf{EF}\,h = \mu Z.h \vee \mathsf{EX}\,Z$ characterizes the set of states satisfying $\mathsf{EF}\,h$ as the least fixpoint of $\tau(Z) = h \vee \mathsf{EX}\,Z$, i.e. as the smallest set of states Z such that $Z = h \vee \mathsf{EX}\,Z$. It is easy to see that all predicate transformers in (2.5) are *monotone*: for any sets $A, B \subseteq S$ with $A \subseteq B$, it is $\tau(A) \subseteq \tau(B)$. Let us denote by $\tau^i(C)$ the i-fold application of τ to the set C, with $\tau^0(C) = C$. Using induction, it follows from monotonicity that $\tau^i(\emptyset) \subseteq \tau^{i+1}(\emptyset)$ for any i:

$$\emptyset = \tau^0(\emptyset) \ \subseteq \ \tau^1(\emptyset) \ \subseteq \ \tau^2(\emptyset) \ \subseteq \ \ldots \ \subseteq \ S\,. \tag{2.6}$$

Since the set of states S is finite, this sequence cannot increase at every stage, i.e. there is an index m such that $\tau^m(\emptyset) = \tau^{m+1}(\emptyset)$. Again by induction it follows that in fact for every $i \geq m$, it is $\tau^i(\emptyset) = \tau^{i+1}(\emptyset)$, i.e. sequence (2.6) converges. It can be shown that the limit is precisely the least fixpoint of operator τ.

The elegance of these easy-to-prove statements is that they are constructive: we can use sequence (2.6) to compute the least fixpoint of τ using a routine that applies τ until no change in value can be observed. This procedure is shown in algorithm 1 (a). An analogous reasoning applies to the greatest fixpoint of any monotone τ; the resulting procedure is shown in (b). The only difference to (a) is that the sequence starts with the full set of states S; transformer τ successively reduces Z until the sequence converges.

The implementation of these algorithms has become known as *symbolic model checking*. We need to be able to efficiently copy sets of states (lines 1 and 3), test

Algorithm 1 Computing fixpoints of a monotone predicate transformer

Input: monotone predicate transformer τ; set of states S

Least fixpoint:

(a)

```
1: $Z := \emptyset$
2: repeat
3:     $Z' := Z$
4:     $Z := \tau(Z)$
5: until $Z = Z'$
6: return $Z$
```

Greatest fixpoint:

(b)

```
1: $Z := S$
2: repeat
3:     $Z' := Z$
4:     $Z := \tau(Z)$
5: until $Z = Z'$
6: return $Z$
```

two sets for equality (line 5), and perform whatever the predicate transformer τ requires. According to the equations in (2.5), this includes set-theoretic and preimage operations. As illustrated in section 2.1.4, all these operations can be applied to binary decision diagrams (BDDs) in time at most linear or low-degree polynomial in the size of the argument BDDs. Specifically, the equality check in line 5 amounts to a BDD equivalence check, and we discussed how to compute predecessors as needed for EX and AX. If the number of BDD nodes is much smaller than the number of elements in a set Z, then these operations can be expected to be cheaper with BDDs than with an explicit data structure for sets, such as lists or even balanced trees.

Complexity. We estimate the complexity of symbolic CTL model checking as a function of the size of the structure $|M|$ and the size of the CTL formula $|f|$. Recall that in the μ-calculus embedding of CTL using fixpoints, the argument Z of the predicate transformer is used only in the form $\mathsf{EX}\,Z$ and $\mathsf{AX}\,Z$, which are simple image computations. It is not used in expressions that themselves require a fixpoint evaluation. For this reason, to evaluate a formula, we can cleanly separate all fixpoints that appear in it, and evaluate them from the inside out, without interleaving. For example, the formula $\mathsf{AG}\,\mathsf{EF}\,P$ ("It is always possible to reach a state satisfying P") has the μ-calculus form

$$\nu Z\,.\,(\mu Y.P \vee \mathsf{EX}\,Y)\ \wedge\ \mathsf{AX}\,Z\,. \tag{2.7}$$

We first compute $Y_0 := \mu Y.P \vee \mathsf{EX}\,Y$. We then substitute Y_0 into equation (2.7) and compute $\nu Z.Y_0 \wedge \mathsf{AX}\,Z$. For each fixpoint calculation, we need at most $|M|$ iterations of the loop in algorithm 1. Overall, the algorithm is linear in $|M|$ and linear in $|f|$. Compare this with the complexity of the LTL model checking algorithm we presented, which is exponential in $|f|$.

2.3.3 Model Checking Tools

To conclude section 2.3, we mention some popular model checkers and their scope, and discuss additional related work.

Mur φ [MD] is an explicit-state verifier for reachability analysis and deadlock detection and as such not a full temporal logic model checker. Mur φ explores a model's state space in a highly optimized fashion and can scan millions of states in a few seconds. Its C-like input language is quite comfortable and includes `if` statements and loops. A program is a collection of *rules*, one of which is nondeterministically chosen in each round and executed. Since Mur φ only analyzes reachability of states and deadlocks, fair scheduling is of no concern. The significance of Mur φ for this dissertation is that it is one of the first serious implementations of *symmetry reduction* (see chapter 4); we return to it in section 8.4.

Spin [Hol97] is an explicit-state LTL model checker. It targets mainly special-purpose software such as asynchronous concurrent systems of processes. Spin's input language, Promela, is a widely-used protocol description notation, with influence on the verification community beyond the Spin model checker.

Spin uses the tableau construction to extract a property automaton from an LTL formula that accepts exactly all paths conforming to the formula. For verification, the goal is to check whether there exists a run of the model automaton that violates the formula. Such a run lies in the intersection of the languages of the model automaton and the complement of the property automaton. In order to avoid the complementation of the property automaton, which can be expensive, Spin solicits the input of a *never claim*: a property that characterizes bad behaviors. Using a tableau, this property is translated into an automaton, which is intersected with the model automaton; the result is checked for emptiness. Any path in the intersection is presented as a counter example.

Spin achieves efficiency beyond straightforward model checking using *partial order reduction* [HP94], a technique that reduces the number of interleavings of execution threads in asynchronous systems. Incidentally, this reduction technique shares some aspects with symmetry reduction—the main focus of this dissertation—, as investigated by A. Emerson, S. Jha and D. Peled [EJP97].

NuSMV [CCB$^+$] is probably the most comprehensive freely available symbolic model checker today. It is a substantial re-implementation of Smv, a model checker implementing the ideas of symbolic model checking developed by K. McMillan [McM93]. Using the fixpoint characterization of CTL, NuSMV is a complete model checker for this logic.

The system modeling language is somewhat restricted, but allows the specification of complex synchronization and coherence protocols. Unlike Mur φ and Spin, however, NuSMV can be used with both synchronous and asynchronous execution models. The model is converted into symbolic form using binary decision diagrams. Like the tool developed as part of this dissertation (chapter 8), NuSMV relies on the Cudd BDD library [Som].

What started as a C++ re-implementation of McMillan's Smv has by now become a giant model checking tool that incorporates many recent developments in the area of formal verification. NuSMV allows the specification of various types of fairness and extends CTL by past-time and real-time temporal operators. Moreover, properties can also be given in LTL.

SAT-based symbolic model checking. A significant addition to NuSMV in recent years was the implementation of symbolic model checking using SAT-checkers, an alternative to BDD-based techniques. Both strategies require the representation of the transition relation of the model as a Boolean formula. Unlike with BDDs, for SAT-based model checking the formula is kept in CNF format. Given a bound k up to which to explore the state space, the transition relation can be unfolded k times, resulting in a (rather large) formula representing all paths of length k from the initial states. Any satisfying assignment to the variables in this formula proves reachability of some condition and can thus be understood as a counter example (of length at most k) to a safety property. To find bugs, the procedure can be repeated with increasing k; but unless a suitable upper bound is known, this method cannot actually prove a safety property correct.

This routine was the first to use SAT methods for symbolic model checking and has become known as *bounded model checking* [BCCZ99]. It was soon thereafter extended by K. McMillan to general—unbounded—model checking. Using quantifier elimination, image operations like $\mathsf{AX}\, g$ can be reduced to a SAT problem. This paves the way for full symbolic CTL model checking; see [McM02] for more details. BDD-based and SAT-based symbolic model checking are often complementary—examples exist for which one representation is exponentially more succinct than the other, and vice versa.

Chapter 3

Abstraction

Overview. In this chapter we provide background on abstraction, a generic term for a collection of techniques to attack model checking's greatest enemy, the state explosion problem. We first introduce its most basic form, existential abstraction, and then derive from this form special cases of abstraction. The different degrees of proximity between the original and the abstract model are discussed.

Abstraction refers to a class of methods to reduce a given model of the system to a smaller one, usually by omitting some detail, such that information relevant for the verification of the property is nevertheless retained. In this context, the original model is called *concrete*, the smaller one—*abstract*. Such methods have one critical additional potential: to obtain a finite-state model from an infinite-state one, thus rendering model checking principally applicable. As a classical example, consider a program with an unbounded integer variable x with initial value 0, and suppose we want to know whether its value can ever become odd. From the straightforward model, which treats x as an integer, we can build an abstract one with a variable $X \in \{even, odd\}$. We assess, for every operation in the program, how it affects the parity of x and update X accordingly. The resulting trivial system of only two global states can now be checked to see whether the state $X = \mathit{odd}$ is reachable.

It is quite easy to turn a given model into a smaller one using a general procedure, as described in section 3.1. The question is, of course, how much resemblance the reduced model bears with the original one, and which properties we can accordingly equivalently verify on the reduced model; this is discussed in section 3.2. Finally, if we find that we cannot verify or falsify the property on the reduced model because it is over-simplified, we have to adjust the abstraction; a popular approach is sketched at the end of the same section.

3.1 Existential Abstraction

Removing detail from a model means to consider states identical that differ only with respect to some apparently unimportant features. Technically, we define an equivalence relation $\equiv$ on the concrete state space; each equivalence class becomes an *abstract state*. There is a transition between two abstract states, i.e. between two equivalence classes $[s]$ and $[t]$, if there exists a concrete transition between some state in $[s]$ and some state in $[t]$:

Definition 5 *Let $M = (S, R, L)$ be a Kripke structure over AP and $\equiv$ an equivalence relation on S. Let $\overline{AP} \subseteq AP$ be the set of atomic propositions that respect $\equiv$.*[1] *The* quotient *of M with respect to $\equiv$ is the structure $\overline{M} = (\overline{S}, \overline{R}, \overline{L})$ (over $\overline{AP}$) with*

$$\overline{S} = \{[s] : s \in S\} \text{ (set of equivalence classes of } \equiv) \tag{3.1}$$

$$\overline{R} = \{([s],[t]) \in \overline{S} \times \overline{S} : \; \exists s_0 \in [s],\, t_0 \in [t] : \; (s_0, t_0) \in R\} \tag{3.2}$$

$$\overline{L}([s]) = L(s) \cap \overline{AP}\,. \tag{3.3}$$

Among the atomic propositions that respect $\equiv$, the labeling function assigns to an equivalence class all those that are true in some and hence in all states in the class.

[1] That is, for every equivalence class $[s]$, all states in $[s]$ agree on propositions from $\overline{AP}$.

Due to the restriction to $\overline{AP}$, the mapping $\overline{L}$ is well-defined.

Let us return to the discussion about integer variable x from above. Figure 3.1 (left) shows a four-state Kripke structure for some program manipulating this variable. A state is labeled *even* if x is even, analogously for *odd*. State u_1 is the initial state. Suppose we want to check whether along all futures, x eventually assumes on odd value: $M, u_1 \models \mathsf{AF}\, \mathit{odd}$. We define an equivalence that relates two states if the parity of x is the same. The quotient structure derived according to definition 5 is shown in figure 3.1 on the right. It has two states $[u]$ and $[v]$ and satisfies the property $\overline{M}, [u] \models \mathsf{AF}\, \mathit{odd}$ since every future from $[u]$ goes through $[v]$. This property is also satisfied by the original structure M. Is this a coincidence, or does M always inherit the verification result from $\overline{M}$? Suppose we now want to check whether along all futures, after one time unit x is invariably odd: $M, u_1 \models \mathsf{AX}\, \mathsf{AG}\, \mathit{odd}$. A quick look at the concrete structure shows that this is indeed the case. There is, however, an abstract path from $[u]$ to $[v]$ and back to $[u]$, which is labeled *even* → *odd* → *even*, so verifying this second property on the quotient would give us a wrong answer.

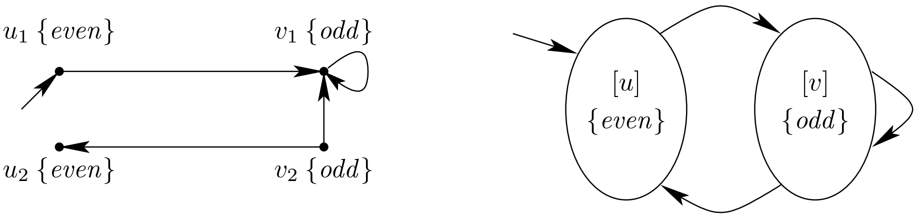


Figure 3.1: A Kripke structure M (left) and an abstraction $\overline{M}$ of it (right)

Most forms of abstraction change the behavior of the program. That is, there may be paths in the abstract model that cannot be mapped to a path in the concrete model, or vice versa. The majority of model checking applications involve checking whether a state violating a certain safety constraint is reachable. For this reason, a fundamental requirement of abstraction techniques is that they be *conservative*:

they must not remove behavior from the system, since this could lead to unsafe states being unreachable and thus go undetected. Instead, abstractions typically add behavior, i.e. transitions, to the model.

This explains what happened in figure 3.1: States v_1 and v_2 were considered equivalent since they are both labeled *odd*. They differ, however, in that from v_2 there is an edge back to a state labeled *even*, while from v_1 there is not. The abstraction, through the edge $[v] \to [u]$, essentially adds to M an edge $v_1 \to u_1$. This addition turned out to be relevant for the second model checking problem above, rendering the abstraction inappropriate. We discuss in section 3.2.2 what we can do in this situation.

3.2 Relationships between Concrete and Abstract Models

In this section we concretize the intuition we obtained from previous examples about relationships between models and their abstractions.

3.2.1 Simulation

Structures M and $\overline{M}$ of the example in section 3.1 do not satisfy the same temporal logic formulas. On the other hand, they are related since every path p through M can be mapped to a path $\bar{p}$ in $\overline{M}$ by mapping each state along p to its equivalence class under $\equiv$, and the labels in corresponding states along the two paths are the same. In particular, each state reachable in M is reachable in $\overline{M}$, in the form of the corresponding equivalence class. We say that $\overline{M}$ has more behaviors than M, or—more technically—it simulates M:

Definition 6 *Let $M_1 = (S_1, R_1, L_1)$ and $M_2 = (S_2, R_2, L_2)$ be two Kripke structures over AP_1 and $AP_2 \subseteq AP_1$, respectively. A relation $\sim \subseteq S_1 \times S_2$ is a* simulation relation *if* $s_1 \sim s_2$ *implies:*

1. $L_1(s_1) \cap AP_2 = L_2(s_2)$*, and*

2. *for every* $t_1 \in S_1$ *such that* $(s_1, t_1) \in R_1$*, there exists* $t_2 \in S_2$ *such that* $t_1 \sim t_2$ *and* $(s_2, t_2) \in R_2$*.*

If $\sim$ *is a simulation relation, we say that* M_2 simulates M_1*.*

(This notion of simulation is distinct from the notion of simulation used in software and hardware testing.) Before we discuss how a simulation relation benefits us, we give some motivating comments. A simulation relation often relates states that agree on "interesting" atomic propositions, but not necessarily on all. Set AP_2 declares which propositions in AP_1 are interesting. The reason for the intersection operator in (1.) is to allow disagreement on other propositions. Regarding (2.), for every successor of s_1 there must be a "corresponding" successor of s_2. This intuition captures the ability of M_2 to simulate M_1.

In practice, we are usually given structure M_1—presumably large—, construct a new structure M_2—presumably smaller—, and prove that M_2 simulates M_1. In fact, we have already seen one such construction:

Theorem 7 *Let* $M = (S, R, L)$ *be a Kripke structure and* $\equiv$ *an equivalence relation on* S*. Let* $\overline{M}$ *be the quotient structure of* M *with respect to* $\equiv$ *(see definition 5). The relation* $\sim\ := \{(s, [s]_\equiv) : s \in S\}$ *is a simulation relation.*

That is, an equivalence relation on S immediately induces a simulating structure— the quotient— that is often smaller, depending on how coarse the equivalence is. For example, structure $\overline{M}$ from figure 3.1 simulates structure M in the same figure.

Given that M_2 simulates M_1, we surmise that every behavior of M_1 is also present in M_2. Consider the sublogic of CTL* that uses only the universal path quantifier; this logic is called ACTL*. If we can prove a formula in ACTL* over M_2, we expect it to hold over M_1, too:

Theorem 8 *Let $M_2 = (S_2, R_2, L_2)$ (over AP_2) simulate $M_1 = (S_1, R_1, L_1)$ (over $AP_1 \supseteq AP_2$) via relation $\sim$, and let f be an ACTL* formula over AP_2. For any $s_1 \in S_1$ and $s_2 \in S_2$ such that $s_1 \sim s_2$, $M_2, s_2 \models f$ implies $M_1, s_1 \models f$.*

Given M_1, once a structure M_2 is found that simulates—and is smaller than—M_1, we attempt to verify f over M_2. Towards this purpose, formula f must be expressed over AP_2. (Thus, when defining M_2 and AP_2, the atomic propositions of f must be included in AP_2.) The theorem says that if the verification attempt succeeds, we can conclude that f is also true over M_1. In the example of the previous section, this was the case with the property $\mathsf{AF}\ \mathit{odd}$.

3.2.2 Abstraction and Refinement

If the verification over the simulating structure M_2 fails, theorem 8 gives no clues. In the example of the previous section, this was the case with the property $\mathsf{AX}\,\mathsf{AG}\ \mathit{odd}$. Fortunately, the model checker can in this case present a counter example (this is always possible for an ACTL* formula), namely the path $[u] \rightarrow [v] \rightarrow [u]$. We must now investigate whether this counter example can be mapped to the concrete system. Without going into details on how to accomplish this, we establish that it cannot: the abstract counter example is *spurious* (unrealizable in the concrete). We say the abstraction is too *coarse*—it must be refined. The abstract counter example tells us that the problem is in $[v]$: the two states it represents are distinct in their ability to lead back to an *even*-labeled state. Guided by this observation, we may decide to split $[v]$ into its two constituents, while keeping the equivalence class $[u]$ intact. On the refined structure, with three states, property $\mathsf{AX}\,\mathsf{AG}\ \mathit{odd}$ evaluates to *true*, as it does on the original structure.

This process is known as (counterexample-guided) *abstraction refinement.* It may turn out that after one refinement, the abstract model still allows spurious counter examples. In this case we may have to refine again. This loop is guaranteed to terminate; in the worst case this happens when the refinement results in the original system, in which case spurious paths are not possible any more. In practice, the number of iterations depends critically on how smart the abstract model is refined; it can be accelerated using human assistance.

3.2.3 Bisimulation

In the previous section we have seen an approach to remedying the problem of oversimplification, by repeated refinement until the model is precise enough to rule out spurious paths. Another option is to avoid this problem from the beginning—by using an exact abstraction. While surprising at first, it is sometimes possible to abstract a model into a smaller one that satisfies exactly the same CTL* properties.

Consider the two structures in figure 3.2 (a). The two B-labeled states in the left structure seem redundant: they have the same label, and their respective predecessors and successors seem to correspond as well. Since CTL* cannot count that the structure on the left has two paths $A \to B \to C$, it cannot distinguish the left structure from that on the right, which has one such path. Formally, the structures are bisimilar, which is nothing but a stronger version of similarity:

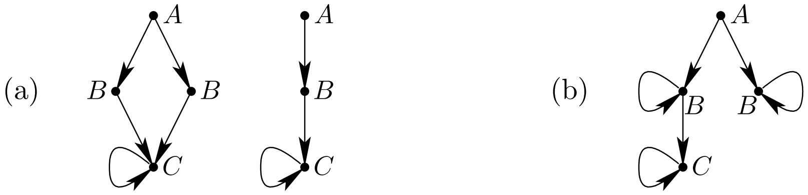


Figure 3.2: (a) Bisimilarity; (b) Simulation-equivalence but not bisimilarity

Definition 9 *Let* $M_1 = (S_1, R_1, L_1)$ *and* $M_2 = (S_2, R_2, L_2)$ *be two Kripke structures over* AP*. A relation* $\approx \subseteq S_1 \times S_2$ *is a* bisimulation relation *if* $s_1 \approx s_2$ *implies:*

1. $L_1(s_1) = L_2(s_2)$,

2. *for every $t_1 \in S_1$ such that $(s_1, t_1) \in R_1$, there exists $t_2 \in S_2$ such that $t_1 \approx t_2$ and $(s_2, t_2) \in R_2$, and*

3. *for every $t_2 \in S_2$ such that $(s_2, t_2) \in R_2$, there exists $t_1 \in S_1$ such that $t_1 \approx t_2$ and $(s_1, t_1) \in R_1$.*

If $\approx$ is a bisimulation relation, we say that M_1 and M_2 are bisimilar*.*

In figure 3.2 (a), the natural bisimulation relation relates states with the same label.

Before we discuss benefits of bisimilarity, some remarks are due. Bisimilarity is a symmetric relation over the set of all structures; simulation is not. Bisimilarity is stronger than simulation. It is, in fact, stronger than simulation in both directions; the latter property is sometimes called *simulation equivalence*. An example of two non-bisimilar structures for which we nevertheless can find two separate relations that show mutual simulation is given in figure 3.2 (b).

The relationship between two bisimilar structures is a strong one:

Theorem 10 *Let M_1 and M_2 (over AP) be bisimilar via relation $\approx$, and let f be a CTL* formula over AP. For any states $s_1 \in S_1$ and $s_2 \in S_2$ such that $s_1 \approx s_2$, $M_1, s_1 \models f$ exactly if $M_2, s_2 \models f$.*

In other words, verification over M_1 produces exactly the same results as verification over M_2 does, for any formula we can express in standard temporal logics. This means that once we have model-checked a formula over either structure (e.g. the smaller one), we never have to worry about spurious paths or refinement. We point out that the benefits of bisimulation differ from those of simulation in two aspects: (i) bisimulation is about any CTL* formula, not just ACTL*, and (ii) bisimulation gives us an "if and only if", not just a result in one direction.

From a more theoretical point of view, theorem 10 means that CTL* cannot distinguish two bisimilar structures. Conversely, if two structures are not bisimilar, one can construct a CTL* formula that distinguishes them. As an example, consider the structures in figure 3.2 (b) and the formula $\mathsf{AX}\,\mathsf{EX}\,C$, which in fact belongs to CTL. With the A-labeled state as start state, this formula is true on the right but not on the left.

Bisimulations are very valuable, but generally hard to come by in practice. Simulation relations are more frequent because they usually allow greater reduction. An exception to this empirical statement is the very topic of this dissertation: Symmetry, which we introduce next, is in part so popular because it allows a bisimilar quotient structure.

Chapter 4

Symmetry and Symmetry Reduction

Overview. In this chapter we lay the foundation for the contributions made by this dissertation. We introduce the concept of symmetry, derive an abstraction mechanism known as symmetry reduction, and also discuss in detail the problems that this reduction has faced in practice, and that we have set to tackle in this dissertation. We conclude the chapter with a short but broad appreciation of symmetry across culture and science.

Symmetry appears virtually everywhere in arts and sciences. Unlike many other ubiquitous phenomena, the characteristics that one intuitively associates with it are surprisingly close to the technical definition we will establish for symmetries of Kripke structures. Namely, an object has symmetry if some aspects of it are *immune to certain transformations.* If we are interested in those aspects only, then for us the object is immune to the change brought about by the transformations.

Consider a concurrent system of many processes, and suppose the processes are all running the same program, just under different names. It appears we may view the processes as interchangeable: a transformation that interchanges them consistently throughout the system may not actually change the system itself. More precisely, it may not change the system's transition relation, i.e. the set of behaviors.

Once symmetry characteristics have been established, we can think about how to exploit it for verification. The idea is that states that are identical up to aforementioned interchanges of processes don't have to be distinguished and can be collapsed into one state of a new reduced system. As for the potential of symmetry reduction, we will see that, given n processes, as many as $n!$ (n factorial) of the original states may be collapsed into a new state. The reduced system is thus exponentially smaller than the original, accounting much for the popularity of symmetry reduction.

4.1 Symmetry of a Kripke Structure

To formalize the intuitive ideas of symmetry, we first define a notion of transformation and how it interchanges processes. Let $M = (S, R, L)$ be a Kripke structure modeling a system of n concurrently executing processes. We model interchanges of the n processes using permutations.

4.1.1 Permutations and Groups

A *permutation* on a set Z is a bijective mapping of Z onto itself. Permutations on a set form a *group* with function composition as the operation. That is, the inverse of a permutation and the sequential composition (product) of two permutations on Z are again a permutation on Z. Further, the identity permutation is its own inverse and is also the neutral element with respect to the group operation. We denote the group of all permutations on a set Z by $\mathit{Sym}\, Z$. The set $\mathit{Sym}\, [1..n]$ has cardinality $n!$ (n factorial).

Permutations acting on $[1..n]$ can be extended to act on the state space S of a Kripke structure. Recall from section 2.1.1 that a state $s \in S$ can be represented as $s = (\vec{v}, s_1, \ldots, s_n)$, where $\vec{v}$ is a vector of global variables and s_i comprises the values of all local variables of process i, collectively known as i's local state. For a permutation $\pi\colon [1..n] \to [1..n]$, we define

$$\pi(s) \;=\; \pi(\vec{v}, s_1, \ldots, s_n) \;=\; \big((\vec{v})^{\pi}, s_{\pi(1)}, \ldots, s_{\pi(n)}\big)\,. \tag{4.1}$$

That is, a permutation acts on a state (i) by acting on the global variables in a way described in the paragraph below, and (ii) by acting on the processes' indices by interchanging their local states. For example, let $s = (A, B, C)$ for a three-process system over local states A, B, C. The left-shift permutation acts on s by left-shifting the local states in s:

$$\pi \;=\; \begin{array}{c|c|c} 1 & 2 & 3 \\ \hline 2 & 3 & 1 \end{array} \quad\Rightarrow\quad \pi(s) \;=\; (B, C, A)\,. \tag{4.2}$$

For global variables, things are a bit more complicated. Some of them are unaffected by a permutation. Consider a binary semaphore that monitors access to a critical code section in a synchronization protocol. The semaphore is set to *true* whenever some process executes the critical code, and is *false* otherwise. A permutation interchanges the processes' local states, but does not affect whether *some* process executes critical code (only *who* does). We call such variables *ID-insensitive*; a permutation acts on them like the identity.

Now revisit the token ring example from figure 2.3 (page 26). The global variable *tok* ranges over process indices: its value is the identity of the one that is allowed to enter its critical section next. We call such variables *ID-sensitive.*

How does a permutation act on an ID-sensitive variable, say v? That is, if i is the value of v in state s, how do we define the value j of v in state $\pi(s)$? Intuitively, the permutation exchanges the rôles of processes i and j. Thus, the local state of process i in state s must be the same as that of process j in state $\pi(s)$, so we have to solve the equation $s_i = s_{\pi(j)}$ for j. The only general solution of this equation is given by $\pi(j) = i$, or equivalently $j = \pi^{-}(i)$. Thus, we define the value of v in state $\pi(s)$ to be $\pi^{-}(i)$.

For example, consider the state $(3, N, T, C)$ of the three-process concurrent system derived from the skeleton in figure 2.3. Processes 1, 2 and 3 are in local states N, T and C, respectively, and *tok* has the value 3. The left-shift permutation π from equation (4.2) changes the state to $(2, T, C, N)$. As a result, the process possessing the token is in local state C, before and after applying the permutation.

It can be shown that with definition (4.1), $\pi \colon S \to S$ is a bijection. In particular, this means that $\pi(S) = S$. How does π affect the transition relation R when we apply it element-wise to the states?

4.1.2 Symmetry

Intuitively, a system of the above form has symmetry if its set of transitions remains invariant when processes are interchanged by certain permutations. Such permutations are called automorphisms:

Definition 11 *An* automorphism *of a structure* $M = (S, R, L)$ *is a permutation* $\pi \colon S \to S$ *such that* $(s, t) \in R$ *implies* $(\pi(s), \pi(t)) \in R$.

That is, applying an automorphism to any transition again results in a valid transition. A permutation acts on a transition by consistently interchanging the components in source and target. Revisiting the example in equation (4.2), suppose the system being modeled allows any process to transit from local state A to local state D, such that $(A, B, C) \to (D, B, C)$ is a valid transition. Applying the left-shift permutation given in the example, we obtain $(B, C, A) \to (B, C, D)$, which must also be a valid transition for the left-shift to be an automorphism.

With this definition in place, symmetry is simply defined as the existence of a (nontrivial) set G of automorphisms. We require that the candidate set G of permutations be a group; the reason is revealed in section 4.2.

Definition 12 *Let G be a group of permutations on $[1..n]$. Structure $M = (S, R, L)$ is* symmetric with respect to G *if every $\pi \in G$ is an automorphism of M.*

Since the automorphisms of a structure themselves form a group, denoted $Aut\, M$, we can rephrase this definition by requiring that G be a subgroup of $Aut\, M$.

We mention some important cases of symmetry. In applications where processes are completely interchangeable, all permutations are automorphisms, so we can choose $G := Sym\,[1..n]$. Such systems are referred to as *fully symmetric.* When processes are arranged in a ring, such as in the dining philosopher's problem, we may be able to rotate the ring without changing the structure. For G, we can choose the group of the n rotation permutations; we speak of *rotational symmetry.* Finally, symmetry groups occurring in practice are often orthogonal products of smaller groups. Consider a solution of the Readers-Writers synchronization problem. In this problem, the participating processes are partitioned into a set of r readers and a set of w writers. Within each set all processes are interchangeable; we can choose $G := Sym\,[1..r] \times Sym\,[r{+}1..r{+}w]$. In general, we would like to recognize as much symmetry as possible and choose the entire automorphism group of M for G. Sometimes, however, the exact group $Aut\, M$ is unknown or expensive to determine; definition 12 only requires G to be a subgroup of it.

We finally observe the following property, which can be concluded from the "groupness" of $Aut\, M$:

Property 13 *If π is an automorphism, then $(s, t) \in R$ exactly if $(\pi(s), \pi(t)) \in R$.*

The condition "$(s, t) \in R$ iff $(\pi(s), \pi(t)) \in R$" can be written as $R = \pi(R)$. Since also $S = \pi(S)$, we obtain the concise characterization $M = \pi(M)$ for an automorphism π.[1]

4.1.3 Detecting and Verifying Symmetry

Before we discuss how we can make use of symmetry towards reducing state explosion, we take a look at how we establish in practice whether a system is symmetric. There are two dimensions along which to consider this problem. One is the traditional "function problem" vs. "decision problem": Is our job to *detect* (quantify) symmetry, without any prior conjecture about the symmetry group, or to *verify* that some given group is a subgroup of $Aut\,M$. The other dimension is the level of abstraction at which the system is considered: a high-level program or a Kripke structure.

Generally, detection and verification of symmetry are expensive when performed at the full Kripke structure level. In addition, we usually want to avoid building the structure up front. An approach that detects symmetry (or violations of it) on the Kripke structure in an on-the-fly fashion is presented in chapter 10.

Most techniques detect or verify symmetry on the level of the input program. The justification for doing so is that an automorphism of the program text—intuitively, a permutation that leaves the program invariant—is also an element of $Aut\,M$ for the induced Kripke structure M; see the discussion of the symmetry principle at the end of chapter 4. A popular way to verify symmetry is to establish syntactic rules for the program and show that every program written in this syntax yields a symmetric Kripke structure. A compiler then verifies that a given program abides by the rules. In chapter 6 we present an example for such an approach. In that chapter, the syntax rules are imposed on the local state transition diagram.

One problem with detecting or verifying symmetry at the program text level is that the Kripke structure may have more symmetries than the program promises.

[1] We can define π to leave the labeling function L of M invariant.

Recalling the discussion of on-the-fly techniques in section 2.1.2, we can weaken the symmetry condition $\pi(R) = R$ to reachable transitions (those with reachable source state) of the system. The program may have statements in it that are never executed; thus the transitions corresponding to such statements can be ignored in the symmetry condition.

In summary, one has to strike a balance between the cost of detecting/verifying symmetry, and the reduction it promises. Choosing a less-than-optimal symmetry group is legal according to definition 12 and may be more efficient than insisting on first finding the full group $Aut\, M$.

4.2 Symmetry Reduction—An Instance of Existential Abstraction

So far we have seen what symmetry is, and how one can recognize it or at least conjecture its presence. In this section we discuss how to exploit it, towards the general goal of reducing the impact of state explosion.

Fortunately, with the background information given in sections 3.1 and 3.2, we have all necessary ingredients at our disposal. We discussed that an equivalence relation on the state space immediately yields a quotient structure that is able to simulate the original structure (theorem 7). Symmetry reduction is an instance of building (in one way or another) this canonical quotient structure. And it is more: we will see that (i) the quotient can be shown to not only simulate the original, but in fact be bisimilar to it, and (ii) due to the way the equivalence is defined, we can make certain estimates about the size of the quotient structure compared with the original.

The equivalence that all builds upon is known as the *orbit relation.* Consider a symmetry group G of a structure $M = (S, R, L)$. For two states $s, t \in S$, we write

$s \equiv t$ if there exists a permutation $\pi \in G$ such that $\pi(s) = t$. In other words, the orbit relation relates two states if they are identical up to the particular arrangement of the processes in them. We have seen the state vectors (D, B, C) and (B, C, D) near definition 11. They are equivalent under the orbit relation provided the left-shift permutation is part of the symmetry group G. The equivalence class—*orbit*—of a state s is the set $[s]_\equiv = \{\pi(s) : \pi \in G\}$.

The requirement that G be a group guarantees that $\equiv$ is an equivalence:

(i). Since G contains the identity permutation, $\equiv$ is reflexive.

(ii). Since G is closed under inversion, $\equiv$ is symmetric.

(iii). Since G is closed under product, $\equiv$ is transitive.

4.2.1 Symmetric Atomic Propositions

An equivalence relation gives us a quotient structure. In order to apply definition 5 (45) and derive a quotient, however, we have to design the set $\bar{AP} \subseteq AP$ of *symmetric* propositions, those that respect the orbit relation $\equiv$. By definition, those are the propositions that all states within an orbit agree on:

$$\bar{AP} \;=\; \{P :\; \forall s \in S,\; \forall \pi \in G :\; P \in L(s) \Leftrightarrow P \in L(\pi(s))\}\,. \tag{4.3}$$

In practice, atomic propositions are usually given as propositional formulas over the local states of the processes (or more generally over simple expressions involving the local variables). In this case, membership in $\bar{AP}$ simply means that the propositional formula is invariant under permutations in G. Typical symmetric atomic propositions quantify over the processes indices, rather than mention any index explicitly. Examples of such propositions include

$$\text{(a)}\;\; \forall i : N_i \qquad\quad \text{(b)}\;\; \exists i : C_i \qquad\quad \text{(c)}\;\; \exists i, j : i \neq j \wedge C_i \wedge C_j\,. \tag{4.4}$$

Atomic proposition (c), for instance, states that no two processes are in their critical section, indicated by local state C. Intuitively, the symmetry of this formula is reflected by the indifference towards the identity of the two processes.

In these examples, the quantifier expressions such as $\exists i$ directly precede the indexed propositional expressions such as C_i, rendering the whole proposition symmetric. This succession is critical. Consider the CTL formula $\mathsf{AG}\,\forall i : (T_i \Rightarrow \mathsf{AF}\,C_i)$, which expresses that whenever a process is in local state T, it will eventually proceed to local state C. In this formula, the indexed propositional expression C_i is also under the scope of a quantifier. The quantifier is, however, separated from C_i by the temporal operator AF. The unquantified expression C_i does not respect the orbit relation $\equiv$ (since different states are labeled with C_1 than with $\pi(C_1) = C_2$ for the flip permutation $\pi = (1 \leftrightarrow 2)$). Also, we cannot push the quantifier $\forall i$ inward in front of C_i, since this changes the semantics of the formula. There is no way to extract symmetric atomic propositions from this formula.

In contrast, the weaker formula $\mathsf{AG}\,\forall i : (T_i \Rightarrow \mathsf{AF}\,\exists j C_j)$ expresses that whenever a process is in local state T, there will eventually be *some* process that proceeds to local state C. Since the expression behind the $\Rightarrow$ operator is independent of i, we can equivalently rewrite this formula as $\mathsf{AG}(\exists i T_i \Rightarrow \mathsf{AF}\,\exists j C_j)$. This time we can choose $\exists j C_j$ as atomic proposition, which does respect $\equiv$; the same holds for $\exists i T_i$.

4.2.2 The Symmetry Quotient

Given the orbit relation, the quotient $\overline{M}$ of M according to definition 5 is in this context known as *symmetry quotient*. By theorem 7, it simulates structure M, via the canonical simulation relation associating states with their orbits. It turns out that the two structures are even bisimilar:

Theorem 14 *Let M be a structure symmetric with respect to a group G. The quotient structure $\overline{M}$, derived from the orbit relation, is bisimilar to M.*

Proof: This theorem, while well known, lies at the heart of symmetry reduction, so we include the short proof. Let $B := \{(s, [s]) : s \in S\}$, and consider a pair $(s, [s])$. We show parts 1 through 3 of definition 9. Property $L(s) = \overline{L}([s])$ holds by construction ($\overline{L}$ was shown to be well-defined). For any t such that $(s, t) \in R$, consider $[t]$. It is $(t, [t]) \in B$ and also $([s], [t]) \in \overline{R}$ by construction (definition 5). Part 3 is more involved. For any orbit $[t]$ such that $([s], [t]) \in \overline{R}$, we know there exist $s_1, t_1 \in S$ such that $(s_1, t_1) \in R$, $s_1 \in [s]$ and $t_1 \in [t]$. From $s_1 \in [s]$ we conclude that there exists $\pi \in G$ such that $\pi(s_1) = s$. Now choose $t_0 := \pi(t_1)$. Since $t_0 \equiv t_1$, it is $t_1 \in [t_0]$. Since also $t_1 \in [t]$, we conclude $[t_0] = [t]$, and therefore $(t_0, [t]) = (t_0, [t_0]) \in B$. Also, $(s, t_0) \in R$: it is $\pi(R) = R$ and $(s_1, t_1) \in R$; applying π to this transition yields (s, t_0). □

Figure 4.1 depicts a small example structure and its quotient. To assess the significance of theorem 14, we recall theorem 10, which tells us in this case that structure M and its symmetry quotient $\overline{M}$ satisfy the same CTL* formulas. Thus, given a verification problem of the form $M, s \models f$ with CTL* formula f over symmetric atomic propositions (equation (4.3)), we can replace it by the equivalent verification problem $\overline{M}, [s] \models f$.

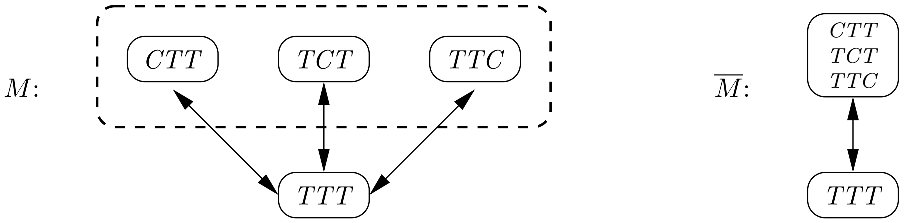


Figure 4.1: An example of a symmetry quotient construction

The motivation for verifying over the quotient is of course that the quotient is smaller, often by much: an orbit can comprise up to $n!$ concrete states. This is the case under full symmetry and for concrete states where the n processes are in pairwise distinct local states; there are $n!$ distinct permutations of these local states.

On the other hand, consider a fully symmetric state, i.e. of the form $(A, \ldots, A)$; its orbit has size 1. The total number of orbits (reachable or not), i.e. the size of the quotient state space, is given by the following property:

Property 15 *For symmetric M over n processes and l local states, the quotient with respect to the orbit relation has* $\begin{pmatrix} n+l-1 \\ n \end{pmatrix}$ *many states.*

This property can be proved using a combinatorial argument. The quantity in property 15 can easily be shown to be at most l^n (the size of the concrete state space), and much smaller than that if n and l get large.

The reduction effect dwindles with decreasing size of the symmetry group. For example, under rotational symmetry, orbits have no more than n members, so we can expect savings of at most a linear factor.

4.3 The Symmetry Quotient in Practice

This section discusses some refinements to the setup of symmetry reduction towards implementing the technique in model checkers. We then show that the construction of the quotient structure faces a fundamental complexity hurdle, which motivates part of the work in this dissertation.

4.3.1 Orbit Representatives

Our description of symmetry reduction so far seems to suggest the following algorithm to verify a system while exploiting its symmetry: (i) build a formal model M, (ii) derive a reduced abstract model $\overline{M}$, (iii) model-check $\overline{M}$. Recall the definition of $\overline{M}$'s transition relation:

$$\overline{R} \;=\; \{([s],[t]) \in \overline{S} \times \overline{S} : \; \exists s_0 \in [s], \, t_0 \in [t] : \; (s_0, t_0) \in R\} \, . \tag{4.5}$$

Although states of the reduced model, such as $[s]$ and $[t]$, are formally defined as equivalence classes, it is unreasonable to encode them this way. Instead, we can define $\overline{S}$ to be a fixed set of *representatives* of each equivalence class. This has the practically useful and simplifying consequence that the abstract states are embedded in the concrete state space: $\overline{S} \subseteq S$; no new modeling scheme is required for the abstract state space. The definition of $\overline{R}$ changes in that the condition $s_0 \in [s]$ in equation (4.5) becomes $s_0 \equiv \overline{s}$. Finally, the quotient labeling function labels a representative with the same atomic propositions as the concrete labeling function does:

$$\begin{aligned}
\overline{S} \quad &= \quad \textit{fixed set of representatives of } {\equiv}\textit{'s equivalence classes} && (4.6)\\
\overline{R} \quad &= \quad \{(\overline{s}, \overline{t}) \in \overline{S} \times \overline{S} :\ \exists s_0 \equiv \overline{s},\, t_0 \equiv \overline{t} :\ (s_0, t_0) \in R\} && (4.7)\\
\overline{L}(\overline{s}) \quad &= \quad L(\overline{s}) \cap \bar{A}P\,. && (4.8)
\end{aligned}$$

A quotient structure defined this way is of course isomorphic to the canonical quotient defined over equivalence classes. We therefore do not need to repeat the conclusions in section 4.2. For the rest of this dissertation, we consider $\overline{M}$ to be defined as above, over representative states.

4.3.2 Detecting State Equivalence

In connection with symmetry reduction, the most important operation is to decide whether two states are equivalent, i.e. identical up to a permutation. We need this operation for example in $s_0 \equiv \overline{s}$ to build the quotient transition relation (equation (4.7)). Under arbitrary symmetries, state equivalence is known to be as hard as the *graph isomorphism problem.* This problem, while considered tractable in practice, causes a lot of overhead for symmetry reduction that reduces its value considerably.

Fortunately, there are important and frequent special cases where state equivalence detection is possible reasonably efficiently. Given the full symmetry group, we can determine whether the vector $(s_1, \ldots, s_n)$ is a permutation of $(t_1, \ldots, t_n)$ by *lexicographically sorting* both and then comparing them for equality. For example, (A, C, A, B) and (B, A, A, C) are equivalent local state vectors because their lexicographically least permutations are both (A, A, B, C). This can certainly be determined in $\mathcal{O}(n \log n)$ time. We note that this method detects whether there is a permutation mapping one state to the other without actually finding this permutation. For a small symmetry group, such as the rotation group, it is possible to just try all permutations to see whether one maps the first vector to the second.

4.3.3 The Orbit Problem of Symbolic Model Checking

For symbolic model checking using BDDs, the state equivalence problem is much worse. To implement $\overline{R}$ as in equation (4.7) symbolically, we need a propositional formula $f(s, \overline{s})$ that detects whether its arguments are symmetry-equivalent. That is, f has the form $f(s_1, \ldots, s_n, \overline{s}_1, \ldots, \overline{s}_n)$ and evaluates to *true* exactly if the vector $(s_1, \ldots, s_n)$ of local states is a permutation (from the group G) of the vector $(\overline{s}_1, \ldots, \overline{s}_n)$.

It turned out that for many symmetry groups, including the important full symmetry group, the binary decision diagram for this formula is of intractable size [CEFJ96]. More precisely, for the standard setting of n processes with l local states each, the BDD for the orbit relation under full symmetry is of size at least $2^{\min\{n,l\}}$. The practical complexity of the orbit relation can be much worse; even if, say, the number of local states l is small, the size of its BDD tends to be intractably large.

As an intuition for this lower bound, consider the related problem of building a finite-state automaton that reads a word of the form $(s_1, \ldots, s_n, \overline{s}_1, \ldots, \overline{s}_n)$ and accepts it exactly if the first n letters are a permutation of the remaining n. One way is to let the automaton memorize, in its states, which of the l^n possible vectors it read until the nth letter, and then compare this information with the remaining n letters. Such memorization requires about l^n different automaton states. Another way is to let the automaton count: While reading the first n letters, the counter for the read letter is increased. While reading the remaining n letters, the corresponding counters are decreased; an attempt to decrease a zero-valued counter means rejection. The set of possible values for all counters must be encoded in about n^l states. Either way, the automaton is of size exponential in one of the parameters.

4.3.4 Ameliorating the Orbit Problem

The motivation for detecting equivalent states is that we want to collapse them in order to reduce the size of the state space. Suppose we were to collapse only some of the equivalent states. This is quickly shown to be legal, since not collapsing some equivalent states does not lose information. Further, we still achieve some reduction. Finally, it may be easier to represent a sub-relation of the orbit relation given by the definition of "some". Intuitively, only few of the potentially $n!$ permutations must be tried in order to determine equivalence: we tolerate some oversights.

This scenario is depicted in figure 4.2 (b). An orbit may now have multiple representatives (black disks), and a single state may be associated with multiple of them. This approach has, accordingly, become known as *multiple representatives*

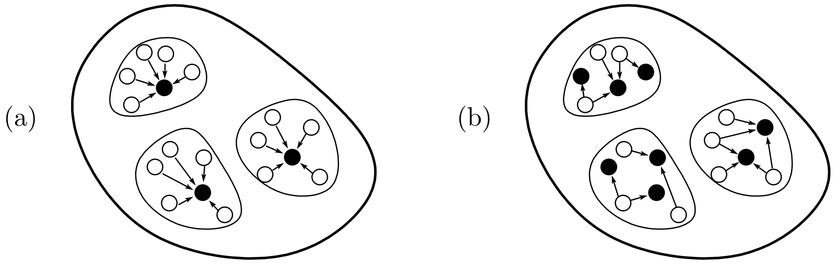


Figure 4.2: Unique (a) or multiple (b) representatives

[CEFJ96]. An obvious disadvantage is that the symmetry reduction effect is negatively impacted by allowing several representatives (i.e. abstract states) per orbit. In fact, for full symmetry, typical choices of the refined equivalence relation make the reduction effect approach a linear factor, down from the original exponential potential of symmetry.

Permitting multiple representatives per orbit is a cosmetic change that ameliorates the orbit problem only to the extent that the benefits of symmetry reduction are diminished. The accomplishment of parts of this dissertation, in particular chapter 5, is to demonstrate that there is a deeper cause for the orbit dilemma, and that we can in fact solve this problem by making more fundamental changes to the way symmetry is exploited.

4.4 Symmetry: A Look Beyond

Symmetry appears virtually everywhere in arts and sciences. In the images of M. C. Escher, geometric symmetry is used as an expression of beauty. Rotational symmetry is abundant in nature, especially in plants, but also in some primitive animals like jelly fish and sea stars. Many higher organisms, including mammals, fish and birds, have reflectional symmetry.[2] Permutation symmetries, as they are used in concurrent systems of processes in this dissertation, occur frequently in mathematics: sums, products and chains of equality are invariant under arbitrary permutations of their arguments. As a result, we find full permutation symmetry in the sine rule and the set of three cosine rules, in binomial rules and many more.

In physics, the *symmetry principle* is a special case of the *equivalence principle* (equivalent causes have equivalent effects). The symmetry principle states that

[2] It has been observed that organisms with reflectional symmetry are generally more advanced than those with rotational symmetry.

in a causal relationship, symmetry in the cause is preserved in the effect. In formal verification, we can view a program text as a cause and the derived Kripke structure as the effect. As discussed before in section 4.1.3, this means that we can predict symmetries of the structure by looking at the program. This is significant since the program text is usually much smaller than the structure.

On the other hand, the effect, the structure, may very well have more symmetries than the cause, the underlying program. This seems to suggest the opposite: to look at the structure directly to find its symmetries, rather than at the program. In practice, this suggestion finds applications in on-the-fly techniques, which generate the (reachable part of the) Kripke structure as the program is simulated. Such techniques suffer less from the size of the structure and may indeed be able to extract, and make us of, more symmetry in the structure than the program seems to promise. An example of such a technique is given in chapter 10.

For a broad and entertaining treatment of symmetry, the reader may wish to consult the books by J. Rosen [Ros75, Ros95].

Part II

Efficient Approaches to Symmetry Reduction

In this part of the dissertation we present principal solutions to the main obstacle in symmetry reduction: the apparent impossibility to efficiently compute and represent the orbit relation. In the previous part we have seen the theory underlying symmetry reduction. It is an instance of existential abstraction, leading to a quotient structure that is bisimilar to the original structure. We have also, however, seen that the quotient is expensive to build, especially with symbolic data structures. The difficulty is that the most fundamental relationship between states, the orbit relation, cannot be represented succinctly as a propositional formula in binary decision diagram format. This is, however, essential in order to build and subsequently model-check the symmetry quotient structure. Due to this deficiency, symmetry reduction was deemed uncombinable with compact data structures and performed mainly with explicit-state model checkers such as MUR φ, SPIN or SMC.

Part II presents solutions to this problem by breaking with the traditional abstraction paradigm of (i) modeling, (ii) reducing, and (iii) checking. Chapter 5 shows an approach that is equivalent in effect to symmetry reduction, but avoids building the reduced model and thus building a representation for the orbit relation. It can be seen as an interleaving of steps (ii) and (iii) above. Chapter 6 demonstrates how a technique that is by itself not new can help us solve the symbolic symmetry reduction problem. This technique performs the reduction step before the modeling, i.e. it reduces the original program text. It is elegant and efficient if applicable, but has a somewhat limited scope in practice. Chapter 7, thus, enhances the technique by cheap but effective analysis methods that precede even the reduction step. We then present DYSYRE, a tool that combines many ideas from this part of the dissertation for efficient symbolic model checking under symmetry. Finally, in chapter 9 we show experimental results obtained with DYSYRE that demonstrate the effect of the developed techniques on many example programs.

Chapter 5

Dynamic Symmetry Reduction

Overview. In this chapter we show an approach to symmetry reduction that avoids building the symmetry-reduced model and thus building a representation for the orbit relation. We first present this approach as an algorithm for symbolic reachability analysis under full symmetry and then generalize it to full CTL model checking given a broader class of symmetry groups.

The approach to be presented can be seen as an interleaving of the reduction and model checking steps. Let us think for a moment about such an interleaving in the context of symbolic models, say using BDDs. In order to leverage the power of BDDs to obtain an entire set of successor states in one step, we must have computed the BDD representation of the transition relation—we cannot build this BDD on the fly. In other words, we have to perform the symbolic modeling step before anything can be done. Folklore seemed to extend this requirement to abstraction: it was believed that with the apparent rigidness of BDDs, abstraction steps cannot be performed on the fly. In our case, such steps include collapsing equivalent states to their representatives.

This chapter breaks with tradition by demonstrating how a compactly represented model can very well be reduced on the fly. We will use BDDs in some unconventional ways that go beyond Boolean or image operations. Not all algorithmic problems can be solved efficiently with compact data structures, but that of detecting equivalent states can.

5.1 Symmetry Reduction without Symmetry Quotient

As mentioned near the end of part I of this dissertation, the straightforward method to apply symmetry reduction seems to be to build a representation of the quotient structure $\overline{M}$ and then model check it. Algorithm 2 (a) shows the standard fixpoint routine (an instance of algorithm 1 (a) on 40) to compute the representative states satisfying $\mathsf{EF}\,\mathit{bad}$, assuming we have a BDD representation of the quotient transition relation $\overline{R}$. We use $\overline{\mathit{bad}}$ to denote the representatives of orbits of *bad* states of M.

Algorithm 2 Two ways to compute the representative states satisfying $\mathsf{EF}\,\mathit{bad}$

(a)

```
$Y := \emptyset$
**repeat**
    $Y' := Y$
    $Y := \overline{bad} \cup \mathsf{EX}_{\overline{R}}\, Y$
**until** $Y = Y'$
**return** $Y$
```

(b)

```
$Z := \emptyset$
**repeat**
    $Z' := Z$
    $Z := \overline{bad} \cup \alpha(\mathsf{EX}_R\, Z)$
**until** $Z = Z'$
**return** $Z$
```

This algorithm is unsuitable for symbolic model checking. The reason is that the construction of the BDD for the quotient transition relation $\overline{R}$ (equation (4.7)) requires the orbit relation, which is of intractable size.

An alternative is to modify the model checking algorithm. Consider the version in algorithm 2 (b). It is identical to (a), except that it uses the operation $\alpha(\mathsf{EX}_R\, Z)$ in the computation of the next iterate: It first applies to Z the backward image operator with respect to R, rather than with respect to $\overline{R}$. It then employs some mechanism α that maps the results to representatives, formally defined as

$$\alpha(T) = \{\bar{t} \in \overline{S} : \ \exists t : t \in T : \ t \equiv \bar{t}\}. \tag{5.1}$$

We can understand algorithm 2 (b) in the context of abstraction in general. The symbol α denotes the *abstraction function*, mapping a set of concrete states to the set of corresponding abstract states. The idea employed by the algorithm is that abstract images, i.e. successors of an abstract state under the quotient transition relation, can be computed *without* the quotient transition relation, using the following three steps: (i) the given abstract state is mapped to the set of concrete states it represents using the *concretization function* γ; (ii) the concrete image is applied to those states; and (iii) the result is mapped back to the abstract domain using α.

What does this procedure look like in our context? Symmetry affords the simplification that γ can be chosen to be the identity function, since abstract states—representatives—are embedded in the concrete state space; they represent themselves. Thus step (i) can be skipped. In algorithm 2 (b)), we can apply EX_R (the concrete backward image operator) directly to the set Z of abstract states, obtaining a set of concrete successor states. Applying α produces the final abstract backward image result.

To concretize this intuition, we prove that the two versions of algorithm 2 compute the same result. We only postulate that α maps the states of its argument set to representatives.

Lemma 16 *Let α be defined as in equation (5.1). Then, for an arbitrary set $\bar{P} \subseteq \overline{S}$ of representatives, $\mathsf{EX}_{\overline{R}}\,\bar{P} = \alpha(\mathsf{EX}_R\,\bar{P})$.*

Proof: In the proof, we slightly overload the symbol α and write $\alpha(t)$ for the unique representative of a single state t, i.e. the unique element of $\alpha(\{t\})$.

$$
\begin{aligned}
& \overline{s} \in \alpha(\mathsf{EX}_R\, \bar{P}) \\
\Leftrightarrow \quad & \langle \text{def. of backward image and function application} \rangle \\
& \exists s, \overline{t} : \overline{s} = \alpha(s) \wedge (s, \overline{t}) \in R \wedge \overline{t} \in \bar{P} \\
\Leftrightarrow \quad & \langle \text{“}\Rightarrow\text{”: } t := \overline{t} \text{ and note } \overline{t} \in \bar{P} \subseteq \overline{S}\text{, so } \overline{t} = \alpha(\overline{t}) = \alpha(t) \rangle \\
& \langle \text{“}\Leftarrow\text{”: } s := \pi(s') \text{ for } \pi : \pi(t) = \overline{t}\text{. Then } \alpha(s') = \alpha(s),\ \pi(s', t) = (s, \overline{t}) \in R \rangle \\
& \exists s', t, \overline{t} : \overline{s} = \alpha(s') \wedge \overline{t} = \alpha(t) \wedge (s', t) \in R \wedge \overline{t} \in \bar{P} \\
\Leftrightarrow \quad & \langle \text{def. of } \overline{R} \rangle \\
& \exists \overline{t} : (\overline{s}, \overline{t}) \in \overline{R} \wedge \overline{t} \in \bar{P} \\
\Leftrightarrow \quad & \langle \text{def. of backward image} \rangle \\
& \overline{s} \in \mathsf{EX}_{\overline{R}}\, \bar{P}. \qquad \square
\end{aligned}
$$

Corollary 17 *The two versions of algorithm 2 return the same set (and they do so with the same number of iterations of the* ***repeat*** *loop).*

Proof: Let Y_i and Z_i denote the ith iterates of the two algorithms. We show by induction that $Y_i = Z_i$ for all i; the two claims of the corollary then follow. It is $Y_0 = \emptyset = Z_0$, and

$$Y_{i+1} = \overline{bad} \cup \mathsf{EX}_{\overline{R}}\, Y_i \overset{\text{(IH)}}{=} \overline{bad} \cup \mathsf{EX}_{\overline{R}}\, Z_i \overset{\text{(L16)}}{=} \overline{bad} \cup \alpha(\mathsf{EX}_R\, Z_i) = Z_{i+1}.$$

Here, (IH) uses the induction hypothesis, and (L16) uses lemma 16. The lemma is applicable since for any i, $Z_i \subseteq \overline{S}$ (by the definitions of $\overline{bad}$ and α). □

Given different implementations of α, algorithm 2 (b) actually represents a family of symmetry reduction algorithms. The definition of α (equation (5.1)) is based on the orbit relation and is therefore inappropriate as a recipe for an algorithm. An alternative is to use the forward image under a precomputed representative relation $\xi = \{(s, r) \in S \times \overline{S} : s \equiv r\}$. This technique was used in [CEFJ96] in connection with multiple representatives; the authors describe ways to obtain such a relation without explicitly using the orbit relation $\equiv$. In contrast, in this chapter we show how to compute the set of representatives of a set of states *dynamically* during the execution of symbolic model checking algorithms, instead of a priori *statically*. This has two advantages:

1. We avoid computing and storing, at any time, the table ξ associating states with representatives, which is expensive.

2. We do not need the complete set of representatives $\overline{S}$, which is required for the computation of ξ. Rather, we only track representatives encountered during the computation.

The algorithm to compute α depends on the type and underlying group of symmetry. In the following section, we first describe in detail the algorithm for the most common and important case of full symmetry. Later, in section 5.5, we present extensions to other symmetries and also generalize the dynamic algorithm to full CTL model checking.

5.2 Computing the Representative Mapping

A scheme for defining representatives frequently used under full component symmetry is the following. Recall that an orbit consists of all states that are identical up to permutations of components, which amounts to permutations of the local states of the processes. Given some total order among the local states, there is a unique state in each orbit where the local states appear in increasing order. This state can be computed by sorting the local state vector of any orbit member.[1]

How can this be accomplished symbolically? Not every sorting algorithm lends itself to symbolic implementation. Compared with an explicit-state algorithm,

[1] We assume for now that there are no symmetry-relevant global variables; section 5.5 generalizes.

instead of sorting one vector of local states, we want to sort an entire set of local state vectors in one fell swoop. One algorithm that allows this efficiently is Bubble Sort, as we motivate in the paragraph below. Bubble Sort is a comparison-based sorting procedure that rearranges the input vector in-place by swapping adjacent out-of-order elements. To symbolically bubble-sort a set of vectors simultaneously, we proceed as follows: Instead of comparing two elements of *the* input vector, the algorithm forms a sub*set* of vectors for which the two elements in question are out of order. Instead of swapping one pair of out-of-order elements, we apply the swap operation to all vectors in the subset, in one step.

The operation of swapping two items turns out to dominate efficiency. Its complexity depends heavily on the distance, in the BDD variable order, of the bits involved in the swap. In order to keep this distance small, we exploit one key feature of Bubble Sort: it is optimal in the *locality* of swap operations—it swaps only adjacent elements. Section 5.3 contains a more detailed efficiency analysis.

Let $\leq$ be a total order on the set of local states. For a fixed global state z, this order induces a total order $\leq_z$ on the set $[1..n]$ of process indices via

$$p \leq_z q \quad \text{iff} \quad z_p \leq z_q. \tag{5.2}$$

Given $\leq_z$, the set of representative states (i.e. states with increasing components) is defined as

$$\overline{S} = \{z : \forall p : p < n : p \leq_z p+1\} = \bigcap_{p<n} \{z : p \leq_z p+1\}. \tag{5.3}$$

For our algorithm, the exact definition of $\leq_z$ is irrelevant; we only need it to be a total order on the local states. This flexibility turns out to be useful in situations where considering just the local states of processes is insufficient to characterize representative states; these situations are discussed in section 5.5. The sorting algorithm looks for states z with components that are not in correct order with respect to $\leq_z$, and swaps them. This is repeated until a fixpoint is reached, see algorithm 3.

Algorithm 3 Computing the representative mapping α using subroutine τ

```
$\alpha(T)$:
1: $Z := T$
2: repeat
3:     $Z' := Z$
4:     $Z := \tau(Z)$
5: until $Z = Z'$
6: return $Z$
```

```
$\tau(Z)$:
1: for $p := 1$ to $n - 1$ do
2:     $Z_{bad} := Z \cap \{z : p >_z p + 1\}$
3:     if $Z_{bad} \neq \emptyset$ then
4:         $Z_{good} := Z \setminus Z_{bad}$
5:         $Z_{swapped} := swap(p, p + 1, Z_{bad})$
6:         $Z := Z_{good} \cup Z_{swapped}$
7: return $Z$
```

For p ranging from 1 to $n - 1$, the predicate transformer τ computes Z_{bad}, the set of states in Z in which components p and $p + 1$ are not in the correct order (line 2 on the right). If Z_{bad} is nonempty, the algorithm first saves the set of states in Z in which p and $p + 1$ *are* in correct order (line 4) and then swaps components p and $p + 1$ in all states in Z_{bad} (line 5). The simultaneous swapping can be achieved by swapping the bits that store components p and $p + 1$ in the BDD for Z_{bad}, which effects all states in Z_{bad}. This is the expensive step of the algorithm; it benefits from these bits being close together (see section 5.3). Finally, the untouched and the swapped states in Z are combined to give the new value for Z (line 6).

5.3 Correctness and Efficiency of the Algorithm

The dynamic algorithm is an instance of algorithm 2 (b). We have already shown more generally that that algorithm computes the same result as algorithm 2 (a). It remains to prove that the implementation of α does what is expected of an abstraction function:

Lemma 18 *Algorithm 3 computes α satisfying equation (5.1).*

Proof: see appendix A.1. □

Corollary 19 *Algorithm 2 (b), using the computation of α in algorithm 3, correctly implements backward reachability analysis on the quotient structure.*

Efficiency Considerations. The set $\{z : p \leq_z p+1\}$, which is by definition $\{z : z_p \leq z_{p+1}\}$, needs to be calculated only once for each p. The condition $z_p \leq z_{p+1}$ can be expressed symbolically with a BDD of size $\mathcal{O}(l^2)$, for the number l of possible local states.

As indicated earlier, the *swap* operation in line 5 of algorithm 3 ($\tau(Z)$) is a bottleneck. In BDD terms, it corresponds to pairwise swapping of all bits that represent the two items to be swapped. The complexity of swapping two bits in all elements of a set Z_{bad}, i.e. computing

$$Z_{swapped} = \{(\ldots x_j \ldots x_i \ldots) : (\ldots x_i \ldots x_j \ldots) \in Z_{bad}\}, \tag{5.4}$$

depends exponentially on the distance d of x_i and x_j in the BDD variable order. To substantiate this claim, we observe that in the BDD for Z_{bad}, every subtree rooted at a node labeled x_i contains at most 2^d nodes labeled x_j. Each such node labeled x_j has an immediate subtree that corresponds to one of the cases affected by the swap, namely $(x_i, x_j) = (0, 1)$ and $(x_i, x_j) = (1, 0)$. These 2^d subtrees must be moved. In the illustration in figure 5.1, these are the $2^2 = 4$ subtrees B, D, E and G.

BDD variable orders usually have the property that it is possible to index the components as $1, \ldots, n$ such that the distance between corresponding bits of components p and q is proportional to $|p - q|$. Consider, for example, the following frequently used orders:

$$\begin{array}{llllllll} \textbf{concatenated:} & b_{11} \ldots b_{1\log l} & b_{21} \ldots b_{2\log l} & \ldots & \ldots & b_{n1} & \ldots b_{n\log l} \\ \textbf{interleaved:} & b_{11} \ldots b_{n1} & b_{12} \ldots b_{n2} & \ldots & \ldots & b_{1\log l} & \ldots b_{n\log l} \end{array} \tag{5.5}$$

where b_{ij} denotes the jth bit of component i. For the concatenated order, the

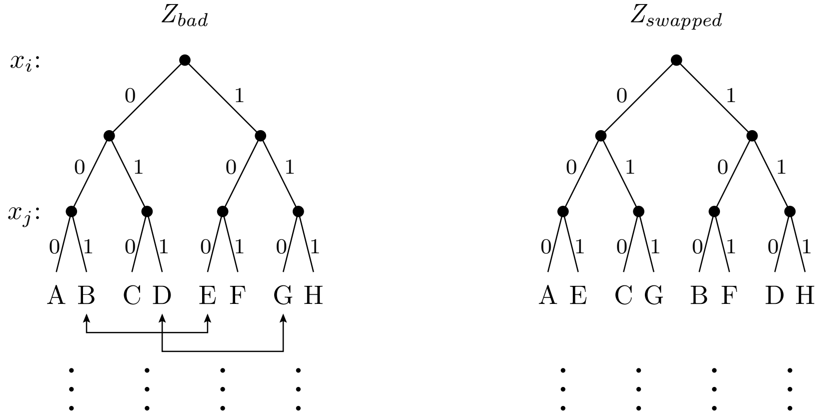


Figure 5.1: Swapping variables in a BDD

distance between the jth bit of component p and the jth bit of component q is $\log l \cdot |p - q|$; for the interleaved order, it is $|p - q|$.

Bubble Sort, among the numerous sorting procedures, enjoys the unique feature of swapping only adjacent components. The distance $|p - q|$ is hence one, for every swap operation, thus minimizing the complexity of swapping. This attests to Bubble Sort's optimality for implementation using BDD.

5.4 Lifting Abstract Error Traces

Suppose the verification of the formula $\mathsf{EF}\, bad$ succeeds; we then have discovered an initial state from which an error state is reachable in the quotient structure. Provided the algorithm has kept a record of the sets of states encountered after each step, an abstract path from the initial to the error state can be reconstructed. Since symmetry reduction is an exact abstraction technique, we can lift this error path back to the concrete system.

The generation of an abstract error path and the lifting to the concrete system can be combined using algorithm 4. The inputs are the initial state i that was backward-reached during the computation of $\mathsf{EF}\, bad$, and the list L of sets of

Algorithm 4 Computing a concrete error path after quotient exploration

```
Input  initial state $i$, set-of-states list $L$
1: $s := i$; $p := (s)$ //$p$ is the path to be constructed
2: for all sets $Z$ in $L$ in reverse order do
3:     $T := \mathsf{Image}_R(s)$ //concrete forward image
4:     $\bar{t} :=$ some element of $\alpha(T) \cap Z$
5:     $s :=$ some element of $Orbit(\bar{t}) \cap T$
6:     push $s$ to the end of $p$
7: return $p$
```

representative states encountered during this computation, excluding the final set (when i was reached). The path construction algorithm itself proceeds forward. It builds a path p as a sequence of states s, beginning with i. After obtaining all successors of s (line 3), a state $\bar{t}$ is extracted from the current iterate Z that represents one such successor in the quotient (line 4). The next state s of the error path is computed in line 5 as a successor of the old s, call it s_{old}, that is represented by $\bar{t}$. The states s_{old} and s satisfy $(s_{old}, s) \in R$ and $\alpha(s) \in Z$.

A slight twist in this algorithm is the computation of the orbit of a state in line 5. This is easy to do if the representative function (or relation) $\xi = \{(s, r) \in S \times \overline{S} : s \equiv r\}$ is available. We mentioned ξ in section 5.1 as a means of computing the abstraction function α, namely as forward image with respect to ξ. Likewise, we can use the preimage with respect to ξ to map a representative state to its orbit:

$$Orbit(\bar{t}) = \{t : (t, \bar{t}) \in \xi\} = \mathsf{PreIm}_{\xi}(\bar{t})\,. \tag{5.6}$$

How do we compute the orbit of a state when ξ is unavailable, such as when performing dynamic symmetry reduction? We can here fall back on the algorithm that was proposed in [CEFJ96] to compute the orbit relation. Starting with the identity relation $\{(s, s) : s \in S\}$, we apply *generators* of the underlying symmetry group to the right hand side of the pairs until a fixpoint is reached. We adjust this approach slightly, as shown in algorithm 5, which computes the orbit of a state $\bar{t}$.

Algorithm 5 Symbolically computing the orbit of a state $\bar{t}$

```
Input  representative state $\bar{t}$
$Z := \{\bar{t}\}$
repeat
    $Z := \{g(z) : z \in Z,\ g$ group generator$\}$
until fixpoint
return $Z$
```

Why do we get away with using a procedure similar to the one used for computing the orbit relation, which we wanted to avoid by all means? The answer is two-fold. First, we apply the above routine only to an individual state $\bar{t}$ found during error path construction, not to a (large) set of states. Second, the result we compute is a set of states, not a relation over states; the latter is much more expensive to deal with symbolically.

5.5 Generalizations

We have so far presented dynamic symmetry reduction for the frequent but special case of reachability analysis under full symmetry. In this section, we generalize it to other types of symmetry and to full CTL model checking. We also demonstrate the impact of symmetry-dependent global variables on the dynamic method.

5.5.1 Other Types of Symmetry

The idea of sorting to obtain unique orbit elements only applies to full component symmetry. We show in this section how to compute α for other, less lucrative, but still somewhat common types of symmetry.

For any symmetry group, a unique representative can be chosen as the lexicographically least member of an orbit. The solution for full symmetry generalizes as follows. Call a symmetry group G of permutations on $[1..n]$ *nice* if there exists a "small" subset F of G with the following property: *A state z is lexicographically* *least in its orbit exactly if there is no* $\pi \in F$ *with* $\pi(z) <_{lex} z$. Given this property, choosing lexicographically least states as representatives is tantamount to defining

$$\overline{S} = \{z : \forall \pi : \pi \in F : \pi(z) \not<_{\text{lex}} z\} = \bigcap_{\pi \in F} \{z : \pi(z) \not<_{\text{lex}} z\} . \tag{5.7}$$

Many common symmetry groups are nice. For full symmetry, F can be chosen as the set of $n-1$ transpositions of i and $i+1$, for $1 \leq i < n$. Set F also happens to be a generating set for the full symmetry group. If G itself is small, $F := G$ is a viable choice. This applies, for example, to the n rotations generated by the left-shift cycle $\varrho := (1\ 2\ \dots\ n)$. Note that here the generating set $\{\varrho\}$ is an invalid choice for F: The vector $z := (BCA)$ is not lexicographically least, yet applying the generating permutation does not make z smaller ($\pi := \varrho^2$ does).

Given a nice group G, consider the algorithm to compute α as before in algorithm 3, but with subroutine τ as shown in algorithm 6. Again, Z_{bad} in line 2 selects the states z in Z that are not lexicographically least. By the niceness of G, this means that for some $\pi \in F$, $\pi(z) <_{\text{lex}} z$. Line 5 applies π element-wise to Z_{bad}. The algorithm for α using the new τ terminates because $<_{\text{lex}}$ is a strict order on the finite set of local state vectors. For partial correctness, we reuse the observations (i) and (ii) from the proof of lemma 18: $\alpha(T) \subseteq \overline{S}$ follows with the same argument, except that the definition of $\overline{S}$ is now given by equation (5.7). Observation (ii) holds without change, and so does the conclusion that $\alpha(T) = \{\bar{t} : \exists t : t \in T : t \equiv_G \bar{t}\}$.

Algorithm 6 Subroutine τ for nice symmetry groups

```
$\tau(Z)$:
  1: for $\pi \in F$ do
  2:     $Z_{bad} := Z \cap \{z : \pi(z) <_{\text{lex}} z\}$
  3:     if $Z_{bad} \neq \emptyset$ then
  4:         $Z_{good} := Z \setminus Z_{bad}$
  5:         $Z_{swapped} := \{\pi(z) : z \in Z_{bad}\}$
  6:         $Z := Z_{good} \cup Z_{swapped}$
  7: return $Z$
```

If G is nice, we expect to have a small set F of permutations to be traversed in line 1. The direct application of π in line 5 may be expensive. Any permutation can, however, be expressed as a product of at most $1/2n(n-1)$ transpositions of *adjacent* elements, often much fewer than that. As argued in section 5.3, these are the least expensive permutations, as for implementation using BDDs. The important point is that algorithm 6 operates in place, and it only swaps neighboring processes, provided the permutations in F are rewritten appropriately.

5.5.2 ID-Sensitive Variables

In section 4.1 we have seen ID-sensitive global variables, which contain process indices such as the identity of a process holding a token. In this case, the condition $\forall p : p < n : z_p \leq z_{p+1}$ is insufficient to guarantee that z is a unique representative state. Consider, for instance, the two states $(1, A, A, B)$ and $(2, A, A, B)$ of a three-process system with one ID-sensitive global variable (listed first). Since components 1 and 2 are both in local state A in both states, the permutation that flips 1 and 2 proves the states symmetry-equivalent. The local states appear in increasing lexicographical order: (A, A, B). Yet, the states differ, compromising uniqueness. The solution is to define the unique representative as the orbit element with increasing local states where the ID-sensitive variables have minimal values (1, in the example above). In this case, $p \leq_z p+1$ means for state z and the local states of p and $p+1$ that either $z_p < z_{p+1}$, or $z_p = z_{p+1}$ and none of the ID-sensitive variables have value $p+1$. This condition is violated for $z := (2, A, A, B)$ with $p := 1$. Thus, the permutation $1 \leftrightarrow 2$ will be applied to z, whereupon it turns into $(1, A, A, B)$.

5.5.3 Full CTL Model Checking

The abstraction mapping α (algorithm 3) was used in section 5.1 towards an efficient strategy to compute $\mathsf{EX}_{\overline{R}}\, Z$ for backward reachability analysis. This algorithm generalizes to all CTL formulas as follows. Recall from section 2.3.2 that existential modalities (EG, EF, EU) have a fixpoint characterization based on existential backward images. For example, $\mathsf{EG}\, f$ can be calculated as the greatest fixpoint of the predicate transformer $\lambda(Z) = f \cap \mathsf{EX}\, Z$. To compute these modalities over the quotient structure, a routine similar to algorithm 2 (b) can be used.

The universal backward image $\mathsf{AX}_{\overline{R}}\, Z$ cannot be replaced by an analogous construct involving α. Suppose we wish to compute the representative states satisfying $\mathsf{AG}\, \mathit{good}$ on the quotient structure. A routine similar to algorithm 2 (a) exists, which computes the greatest fixpoint of $\lambda(Z) = \overline{\mathit{good}} \cap \mathsf{AX}_{\overline{R}}\, Z$. In general, however, $\alpha(\mathsf{AX}_R\, Z) \subsetneqq \mathsf{AX}_{\overline{R}}\, Z$. The underlying problem is that the abstraction function α distributes over set union, but not intersection:

$$\begin{array}{rcrcll} \alpha(P \cup Q) & = & \alpha(P) & \cup & \alpha(Q), & \quad \text{but} \\ \alpha(P \cap Q) & \subsetneqq & \alpha(P) & \cap & \alpha(Q) & \quad \text{(in general).} \end{array} \tag{5.8}$$

The solution is to reduce universal to existential modalities. Care must be taken in that over the quotient structure, negation corresponds to complementation with respect to $\overline{S}$, the set of representatives:

$$\mathsf{AX}_{\overline{R}}\, Z = \neg(\mathsf{EX}_{\overline{R}} \neg Z) = \overline{S} \setminus \mathsf{EX}_{\overline{R}}(\overline{S} \setminus Z) = \overline{S} \setminus \alpha(\mathsf{EX}_R(\overline{S} \setminus Z)). \tag{5.9}$$

This solution requires the set $\overline{S}$ of all representatives. Depending on the application and the definition of representatives, the BDD for this set can be (but is not always) large. It can be computed as $\alpha(\mathit{true})$, but the direct way based on the expression $\bigcap_{p<n} p \leq_z p+1$ is often more efficient. In section 5.5.4, we discuss situations in which the computation of $\overline{S}$ can be avoided. Other than $\overline{S}$, the above equations only involve Boolean primitives, existential backward image with respect to R, and the abstraction function α. This makes the dynamic technique complete for CTL.

5.5.4 Computing Representative Sets for Atomic Propositions

The mapping α can be used to convert atomic propositions into their representative form, such as *bad* into $\alpha(\mathit{bad}) = \overline{\mathit{bad}}$ in order to compute $\mathsf{EF}_{\overline{R}}\,\overline{\mathit{bad}}$. Since the set of *bad* states is, per symmetry requirement, invariant under permutations, it contains all representatives of *bad* states, making it a superset of $\overline{\mathit{bad}}$. Thus, it seems that we can "conservatively" replace $\overline{\mathit{bad}}$ by *bad* as the starting point for backward search, saving us one application of α. This would be beneficial when we need the negation of an atomic proposition. Suppose instead of the set of *bad* states, an application defines what *good* states are. Since computing $\mathsf{AG}_{\overline{R}}\,\overline{\mathit{good}}$ requires the potentially unwieldy set $\overline{S}$ of representative states (see section 5.5.3), we want to compute $\mathsf{EF}_{\overline{R}}\,\overline{\mathit{bad}}$ instead. Unfortunately, $\overline{\mathit{bad}}$ must be computed as $\overline{S} \setminus \overline{\mathit{good}}$ (negation in the quotient) and thus requires $\overline{S}$ as well. In this situation, we would like to simply run algorithm 2 (b) with $\overline{\mathit{bad}}$ replaced by $\mathit{bad} = \neg \mathit{good}$, avoiding the computation of $\overline{S}$ altogether.

The justification for this replacement is as follows. Consider two sets of states Z and V that are identical up to permutations, i.e. $\mathit{Orbit}(Z) = \mathit{Orbit}(V)$. Such sets are indistinguishable in the quotient; in particular, quotient image operators yield the same result when applied to Z and to V:

Theorem 20 *For Z and V with $\mathit{Orbit}(Z) = \mathit{Orbit}(V)$, $\alpha(\mathsf{EX}_R\, Z) = \alpha(\mathsf{EX}_R\, V)$.*

Proof: We show "$\subseteq$"; the other direction follows by commutation.

Assume $\bar{t} \in \alpha(\mathsf{EX}_R\, Z)$, i.e. $\bar{t}$ is the representative for some $t \in \mathsf{EX}_R\, Z$. Thus there is an element $z \in Z$ with $(t, z) \in R$. Let $\bar{z}$ be the representative state of z, then $\bar{z} \in \mathit{Orbit}(Z) = \mathit{Orbit}(V)$. Hence, there is an element $v \in V$ symmetry-equivalent to $\bar{z}$ and thus to z, say $v = \sigma(z)$. By symmetry of R, we can apply σ to (t, z) to obtain $(\sigma(t), \sigma(z)) = (\sigma(t), v) \in R$, thus $\sigma(t) \in \mathsf{EX}_R\, V$ and $\bar{t} \in \alpha(\mathsf{EX}_R\, V)$. □

Suppose now we replace $\overline{\mathit{bad}}$ by *bad* in algorithm 2 (b). Let Z_i be the ith iterate of the original algorithm, and V_i be the ith iterate after the replacement. An easy induction proof shows that for all i, $\mathit{Orbit}(Z_i) = \mathit{Orbit}(V_i)$ (enabling theorem 20), $Z_i \subseteq V_i$, and $V_i \setminus Z_i \subseteq \mathit{bad} \setminus \overline{\mathit{bad}}$. Thus, the algorithm after the replacement produces the same result as the original algorithm 2 (b), except for some additional elements from $\mathit{bad} \setminus \overline{\mathit{bad}}$. A similar result can be shown for the $\mathsf{AX}_{\overline{R}}$ operator, computed as in equation (5.9).

The additional, non-representative states must be taken into account when interpreting the result, which often amounts to checking whether the computed set of states contains any initial states, such as checking whether $\overline{\mathit{init}} \cap \mathsf{EF}_{\overline{R}}\, \overline{\mathit{bad}}$ is nonempty. Fortunately, for two sets of representative states $\bar{I}$ and $\bar{B}$ and over-approximations $\tilde{I} \supseteq \bar{I}$ and $\tilde{B} \supseteq \bar{B}$ that do not contribute any new orbits,

$$\tilde{I} \cap \tilde{B} = \emptyset \quad \text{iff} \quad \bar{I} \cap \bar{B} = \emptyset\,. \tag{5.10}$$

Thus, for backward reachability analysis, we can start from bad instead of $\overline{\mathit{bad}}$ and compare the backward-reachable states against init instead of $\overline{\mathit{init}}$. This optimization avoids having to compute the set of representative states $\overline{S}$ and still achieves an exact verification result. We found that the overhead by carrying around extra states (from $\mathit{bad} \setminus \overline{\mathit{bad}}$) is not noticeable.

5.6 Conclusions and Bibliographic Notes

Dynamic symmetry reduction is a symbolic abstraction technique that avoids precomputing the abstraction function by instead using an efficient symbolic algorithm to map concrete to abstract states on the fly. As such, it also benefits from generating only reachable abstract states, which may be few if an error is detected soon after the start of the exploration. We present experimental evidence for these claims in chapter 9, where we compare the dynamic with other symmetry reduction techniques.

Bubble Sort is usually regarded simple-minded and inept for large sorting problems. However, the efficiency of an automated solution to a problem is always a synergy between the algorithm and the data structure. In our case, the main expectation of the data structure—BDD—is that it be concise. This feature of BDDs (over extensional representations) is paid for by forgoing efficiency of certain operations that are otherwise considered elementary, such as arbitrary exchanges of elements in an array. We believe the locality of Bubble Sort to be paramount, i.e. its affecting only nearby elements and being in-place.

Dynamic symmetry reduction was first presented in [EW05] and compares with other work as follows. E. Clarke, R. Enders, T. Filkorn and S. Jha proposed the admission of *multiple* orbit representatives [CEFJ96] to alleviate the orbit problem. We discussed this approach briefly in section 4.3.3. It affords the possibility to map a state to that representative of its orbit for which this mapping is most efficient. The relation, call it ξ, associating a state with all of its potential representatives, is pre-computed in a BDD. This method, albeit an improvement, is ineffective for systems of interesting size. This is in part because the BDD for ξ is generally still huge, and in part because of the multiplicity of the representatives, such that symmetry is not exploited to the fullest extent. In comparison, the method presented in this chapter computes representatives of states dynamically, embedded in the model checking process. This has the important advantage that there is no need to compute, let alone store for the lifetime of the program, the representative mapping ξ. Further, we only maintain representatives actually encountered during the computation, which might be few. In contrast, pre-computing all representatives may consume a lot of resources, only to find during model checking that a state close to an initial state already has a bug. As an added benefit, the dynamic solution preserves the uniqueness of orbit representatives.

Another, and very different, attempt to implement symmetry reduction for symbolic model checking is due to S. Barner and O. Grumberg [BG02]. Their approach mainly targets falsification, i.e. (like with testing) discovering the presence of errors, instead of proving their absence. If too large, the set of reached representatives is under-approximated, which renders the algorithm inexact. Also, this work employs multiple representatives, forgoing some of the achievable compression.

A technique to apply symmetry reduction in SAT-based model checking—another form of symbolic reasoning—was proposed by D. Tang, S. Malik, A. Gupta and N. Ip [TMGI05], incidentally in the same year as, but a few months later than, dynamic symmetry reduction. The authors exploit that in each round of bounded model checking (see section 2.3.3), only the final state of a potential path to an error needs to be constrained to be a representative. Since intermediate sets of states are never explicitly enumerated, but represented implicitly in the unrolled formula for the transition relation, the potential for state explosion is shifted from the state space to the SAT-solving algorithm. SAT-based and BDD-based model checking techniques are generally considered complementary, which is why there is a justification for symmetry reduction (or most any abstraction technique, for that matter) in both domains. For the same reason, the dynamic technique and symmetry reduction for SAT-based model checking are hard to compare since the differences in the results are blurred by the different conciseness of the model representation (CNF vs. BDD).

Dynamic symmetry reduction switches back and forth between the concrete state space S and the abstract state space $\overline{S}$ of representatives. The theory that establishes the relationship between these two domains is known as *abstract interpretation* and goes back to P. and R. Cousot [CC77]. That paper also formalizes the notions of the abstraction and concretization functions α and γ.

Chapter 6

Symmetry Reduction with Generic Representatives

Overview. In this chapter we present an efficient and elegant but *demanding* alternative method for circumventing the orbit problem in symbolic model checking. It involves a translation of the input program into one whose Kripke structure contains the symmetry quotient of the Kripke structure of the original program as an isomorphic embedding. Since this translation can be complicated, we begin by presenting this process in detail for a simple example program. We then describe the formal procedure to translate the input program and show that the Kripke structure derived from the result can be represented compactly as a BDD.

The following two chapters are about fully symmetric systems, or systems orthogonally composed of fully symmetric subsystems. Thus, when we speak of symmetry, we always mean with respect to the full symmetry group.

6.1 Quotient Structure Revisited

Full symmetry, where the symmetry group contains all permutations on $[1..n]$, occurs quite frequently in practice, whenever a system is composed of unordered, pairwise interchangeable components. This is the case for clique networks of processes, but also for star topologies where components communicate via a centralized hub, such as in some cache coherence protocols. In the latter cases, the hub can be "factored out": instead of treating it as a privileged process that destroys the symmetry, we can store its local state in global variables. The remaining processes then are indistinguishable, such that full symmetry reduction can be applied to them.

The basic operation in symmetry reduction is and remains to detect whether two states are equivalent. In general, this means to check whether there exists a permutation in the group that maps one state to the other. In the case of full symmetry, the group equals $\mathit{Sym}\,[1..n]$, so the question becomes whether the vector of local states in one process is any permutation of that of the other.

A permutation, viewed as an operation on a finite sequence of objects, rearranges those objects, but does not change the number of objects of any given type in the sequence. In our case, this means that for any state s and a permutation $\pi(s)$ of its local state vector, the number of occurrences of each local state (= type) is the same. The key observation for the technique presented in this chapter is that if all permutations are eligible, the inverse of the above statement is also true. That is, two states s and t that satisfy the following condition are equivalent: for any existing local state L the number of occurrences of L in s and t is the same.

The consequence of this observation is that under full symmetry, an orbit is precisely characterized by the number of occurrences of L, for each existing local state L. As an example, consider the following orbit under full symmetry:

$$\begin{array}{cccc} (A, A, B, C) & (A, A, C, B) & (A, B, A, C) & (A, B, C, A) \\ (A, C, A, B) & (A, C, B, A) & (B, A, A, C) & (B, A, C, A) \\ (B, C, A, A) & (C, A, A, B) & (C, A, B, A) & (C, B, A, A) \end{array} \tag{6.1}$$

It is characterized by two occurrences of A, one of B and one of C (and none of all others, if any). We can succinctly write this orbit as the counter tuple $(2, 1, 1)$. Such a tuple is called a *generic representative* in [ET99]: it represents the orbit just like a traditional representative does, but it is generic in the sense that it does not single out any specific orbit member.

This idea can be used to build a representation of the symmetry quotient. The quotient state space is the set of generic representatives, which is the set of all counter tuples of dimension l: for each of the l local states we have to remember the number of processes residing in it. Thus, we define $\overline{S} = [0..n]^l$. The representation of $\overline{R}$ is more involved. For an explicit-state model, we could find successors of a generic representative by mapping it back to an arbitrary member of the orbit that it represents, finding successors of that member using the concrete transition relation R, and map the result back to the generic state space. For a BDD-based symbolic model, however, this procedure is infeasible since the generic state space cannot be embedded into the specific state space—the spaces are disjoint.

The solution is to translate the given program text *before* building a model. That is, the program P, whose behavior is given by local state changes of processes and perhaps by assignments to global variables, is rewritten into an equivalent program $\widehat{\mathcal{P}}$ whose behavior is given by updates to global counter variables and perhaps by assignments to other global variables. If successful, the reachable part of the new program's Kripke structure $\widehat{M}$ is isomorphic to the quotient $\overline{M}$ of the Kripke structure M of the original program P. The advantages are evident:

1. We never need the orbit relation, since the reduction step is not performed on the Kripke structure of the program, but on the program text itself.

2. We never build the Kripke structure of the original program (unlike with dynamic symmetry reduction).

3. We don't need to use a special algorithm for model checking under symmetry; standard model checking can be applied to $\widehat{M}$ (unlike with dynamic symmetry reduction). In particular, we never need to check equivalence of states.

Achieving the translation is, however, more complicated than it may appear. We therefore give an example for a simple program in the next section before we describe the general procedure (section 6.3). We assume programs are given in "preprocessed" form as synchronization skeletons. That is, the behavior of the processes is represented abstractly in terms of local state changes. Global variables can be used in guards and can be assigned as usual. Both parts—processes and global variables—of the skeleton contribute to the difficulty of the translation process, in different ways. Global variables pose an algorithmic challenge, as exhibited in the example below and detailed in the formal translation procedure. In chapter 7, we address the problem of extracting a synchronization skeleton from a conventional program, and the care that must be taken in order to avoid a particular flavor of state explosion that can accompany counter abstraction.

6.2 Counter-Abstracting Symmetric Programs—An Example

Consider a token-ring solution to the n-process Mutual Exclusion problem with a global variable $tok \in [1..n]$, as depicted in figure 6.1. The skeleton allows a process to enter its critical section C if it currently possesses the token ($tok = i$). Upon

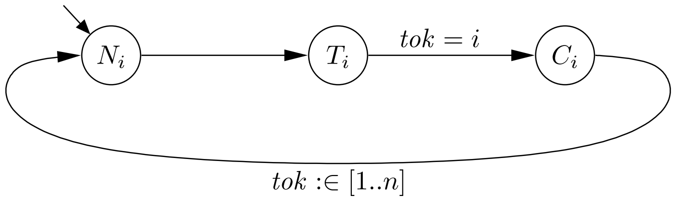


Figure 6.1: Skeleton for a token-ring solution to the Mutual Exclusion problem

leaving C, it sets *tok* to a nondeterministic value in $[1..n]$. The skeleton induces a concurrent system in which the processes asynchronously execute local transitions and update variable *tok*, depending on their current local state and satisfaction of the transition guards. The Kripke structure of this system is fully symmetric, as we see in the next section for skeletons written in a specific input syntax. For this example, we content with the hint that the skeleton does not refer to any process identifier other than that of the executing process (i).

We now want to construct a new program based on counters. Instead of a local state variable for each process, we conversely declare global counter variables for each local state, calling them n_N, n_T, n_C. A slight challenge is provided by the *tok* variable with range $[1..n]$. Since the counter variables deliberately ignore process identities, we cannot check a guard like $tok = i$ any more. Recall, however, the discussion about ID-sensitive global variables, such as *tok*, from section 4.1. The motivation for the definition of a permutation action on such variables was that the local state of the process that the variable points to must not change. For example, the states $(tok = 2, N, T, C)$ and $(tok = 3, N, C, T)$ are equivalent, which should be reflected in the counter program to be built. This observation comes to the rescue in the translation effort: Instead of the variable $tok \in [1..n]$, we create a new variable *TOK* that records just the local state of the token process. Thus, *TOK* ranges over $\{N, T, C\}$.

The translated program consists of the variables and statements shown in figure 6.2. The values of the counter variables range from zero to the number of

```
//**Variables:**
$n_N, n_T, n_C$   :   $[0..n]$
$TOK$             :   $\{N, T, C\}$

//**Initial values:**
$(n_N, n_T, n_C)$   :=   $(n, 0, 0)$
$TOK$               :=   $N$
```

```
//$N \to T$:
**if** $n_N > 0$ :
  **if** $TOK = N$ :
    **if** $n_N = 1$ :
      $TOK := T$
    **else**
      $TOK :\in \{N, T\}$
    **end if**
  **end if**
  $n_N := n_N - 1$
  $n_T := n_T + 1$
**end if**
```

```
//$T \overset{tok=i}{\longrightarrow} C$:
**if** $n_T > 0 \wedge TOK = T$ :
  $TOK := C$
  $n_T := n_T - 1$
  $n_C := n_C + 1$
**end if**
```

```
//$C \to N$, $tok :\in [1..n]$:
**if** $n_C > 0$ :
  $n_C := n_C - 1$
  $n_N := n_N + 1$
  $TOK :\in \{L : n_L > 0\}$
**end if**
```

Figure 6.2: Generic version of the token-ring solution to the Mutual Exclusion problem

processes, n. The initial values of the counter variables and that of variable TOK follow from all processes starting out in local state N.

All transitions in the new program require that the counter of the source local state is positive, since the transition can be taken only if there is a process in that local state. The first transition, $N \to T$, has apparently nothing to do with the token, since *tok* does not explicitly appear in it. The process executing it, however, might be the one possessing the token, in which case the new variable TOK must be updated from N to T. If $TOK = N$ and $n_N = 1$, we know the executing process has the token, and we set TOK to T. If $TOK = N$ and $n_N > 1$, then the process executing the transition may or may not be the one possessing the token, so we must set TOK to T, or TOK must remain at N, respectively. Hence, the new program has two transitions in this case, which we abbreviate by a nondeterministic assignment $TOK :\in \{N, T\}$. Finally, the actual local state change is reflected by decreasing n_N and increasing n_T. A similar but simpler reasoning motivates the translation of the other two transitions. In the last statement, the condition $n_L > 0$

in the nondeterministic assignment to TOK ensures that only local states in which at least one process resides are considered.

The property to be verified also needs to be translated into counters. As an example, compare the mutual exclusion (safety) and communal progress (liveness) requirements in specific and generic notation:

	specific	generic
Safety:	$\mathsf{AG}\,\forall i, j : i \neq j : \neg(C_i \wedge C_j)$	$\mathsf{AG}(n_C < 2)$
Liveness:	$\mathsf{AG}(\exists i T_i \Rightarrow \mathsf{AF}\,\exists j C_j)$	$\mathsf{AG}(n_T > 0 \Rightarrow \mathsf{AF}\,n_C > 0)$.

The liveness property states that if there is *some* process in its trying region, then in any possible future, there should eventually be *some* process entering its critical section. This property is weaker than progress of an individual process, formally $\mathsf{AG}\,\forall i : (T_i \Rightarrow \mathsf{AF}\,C_i)$. The latter formula is, however, asymmetric, as we have seen in section 4.2. It can therefore not directly be verified over a symmetry-reduced structure. One approach to overcoming this problem is to factor out one of the processes and treat its local variables as global. The progress property is formulated for this process, and symmetry reduction is applied to the remaining ones. This approach is described in more detail by A. Pnueli, J. Xu, and L. Zuck [PXZ02], incidentally for counter-abstracted programs.

To see that implementing the above translation is tantamount to performing symmetry reduction on the program text, notice that all states from one equivalence class of the original system are mapped by the translation to the same tuple (TOK, n_N, n_T, n_C) over counters. This tuple can therefore be viewed as an "unusual notation" for the representative of the orbit—we adopt the term *generic representative* coined in [ET99]. The new program can now be transformed into a Kripke structure, represented by BDDs and model-checked, without any further consideration of symmetry.

6.3 Formalizing the Translation Process

In this section we present a general procedure that translates a program given as a synchronization skeleton into an equivalent counter program. We specify the requirements that the skeleton must satisfy in order for the translation to make sense. We derive a Kripke structure from the skeleton and finally formalize the notion of equivalence between the programs via equivalence between the Kripke structures they induce. ID-sensitive global variables like *tok* play a particular rôle during the translation process, as the example in the previous section has indicated.

A structure induced by a synchronization skeleton is a promising candidate for symmetry, since all processes execute the same parameterized program. This fact alone, however, is insufficient: guards and actions on local state transitions can depend on the identity of the executing process in a way that limits or destroys the otherwise apparent symmetry. For instance, the action $tok := (tok \pmod{n}) + 1$ of the skeleton in figure 2.3 is intuitively invariant only under the n rotation permutations. To ensure full symmetry of the Kripke structure to be derived later, we have to place conditions on the synchronization skeleton syntax.

6.3.1 Input Program Syntax

We assume a program $\mathcal{P}$ in the form of the following parameters: (1) the number n of processes, (2) any number of ID-insensitive global variables, given as a single vector $\vec{v}$ with range V (cross product of individual ranges), (3) any number z of ID-sensitive global variables, given as $\vec{d} = (d_1, \ldots, d_z)$ with range $[1..n]^z$, and (4) a synchronization skeleton, parameterized by i. The latter is a finite directed graph, each node of which represents (and is identified with) a local process state; call their number l. The edges may be labeled with a guard and an action (which default to *true* and *no-op*, respectively).

Syntax of guards. Guards are arbitrary Boolean combinations of *basic guards*, which in turn are expressions over local states and global variables. Basic guards over local states are required to be fully symmetric:

Definition 21 *For a quantified Boolean formula h over atoms of the form L_i, $i \in [1..n]$, and a permutation π on $[1..n]$, define $\pi(h)$ by π acting upon the indices. Formula h is* fully symmetric *if for every π, $h \Leftrightarrow \pi(h)$ is a tautology.*

Some basic guards satisfying this definition are listed in table 6.1. This table also shows expressions over local state counters for these guards, as we will use them later for the translation. As an example, the guard *exactly one process is in local state* L, formally $(\exists i : L_i) \wedge (\forall i, j : L_i \wedge L_j \Rightarrow i = j)$, is equivalent to the conjunction of the negation of basic guards 1 and 3 from the table. It is more succinctly written as $n_L = 1$ over counter variables.

no.	Basic Guard	Counter version	Meaning
1	$\forall i : \neg L_i$	$n_L = 0$	none
2	$\forall i : L_i$	$n_L = n$	all
3	$\exists i, j : i \neq j : L_i \wedge L_j$	$n_L \geq 2$	at least two

Table 6.1: Fully symmetric basic guards on local states

Any (syntactically valid) expression over ID-insensitive global variables is by nature fully symmetric and thus a legal basic guard. As for an ID-sensitive variable d, we allow the expressions $d = i$ and $d \neq i$ as basic guards (recall that i is the parameter of the synchronization skeleton).

Syntax of actions. An action consists of at most one assignment to each of the global variables. The semantic model for the assignments—e.g. parallel or sequential—is left to the implementation, since it is irrelevant for the translation of the source program into generic representatives.

As with guards, to ensure full symmetry the syntax of actions must be restricted. Any (syntactically valid) assignment to the ID-insensitive variables is legal, since it does not affect the symmetry of the program. For an ID-sensitive variable d we allow the following three types:

$$d := i \qquad d :\in [1..n] \qquad d :\in ([1..n] \setminus \{i\})\,. \tag{6.2}$$

The last two actions intuitively assign a nondeterministic value in the given set to d.

The asynchronous execution semantics of such a program is given by the derivation of a Kripke structure:

Definition 22 *A program specified in the above syntax defines a Kripke structure $M = (S, R, L)$ as follows: $S = V \times [1..n]^z \times [1..l]^n$, and R contains all pairs (s, t) with*

$$s = (\vec{x}, \vec{k}, s_1, \ldots, s_{i-1}, X, s_{i+1}, \ldots, s_n), \quad t = (\vec{x}', \vec{k}', s_1, \ldots, s_{i-1}, Y, s_{i+1}, \ldots, s_n)$$

such that there is an edge $e\colon X \to Y$ in the skeleton with a guard that evaluates to true for $\vec{v} = \vec{x}$, $\vec{d} = \vec{k}$ and local states as in s, and e's action $\mathcal{A}$ satisfies the Hoare triple $\langle \vec{v} = \vec{x} \wedge \vec{d} = \vec{k} \rangle \mathcal{A} \langle \vec{v} = \vec{x}' \wedge \vec{d} = \vec{k}' \rangle$. Finally, L labels a state with fully symmetric expressions (over local state variables and the ID-insensitive global variables) that are true at that state.

The following theorem shows that symmetry reduction can be applied to M.

Theorem 23 *Given the permutation action*

$$\begin{array}{rcl} s & = & (\vec{x}, k_1, \ldots, k_z, s_1, \ldots, s_n) \\ \Rightarrow \quad \pi(s) & = & (\vec{x}, \pi^{-}(k_1), \ldots, \pi^{-}(k_z), s_{\pi(1)}, \ldots, s_{\pi(n)})\,, \end{array} \tag{6.3}$$

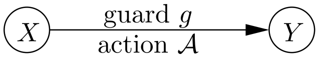
structure M is fully symmetric, i.e. $\pi(M) = M$ for every π.

Proof (sketch): We have to show for an arbitrary permutation π that π is an automorphism of M. By definition 11, this means that for any two states s and t,

$(s, t) \in R$ implies $(\pi(s), \pi(t)) \in R$. To accomplish this, we apply definition 22 to (s, t) and conclude that there is an edge $e = A \to B$ in the skeleton and an index i such that the local state change of process i from A to B in global state s accounts for the transition (s, t). We have to again find an edge e' and an index i' that account for the transition $(\pi(s), \pi(t))$. We choose $e' := e$ and $i' := \pi^{-}(i)$. A case analysis over the allowed guards and actions attached to e shows the legitimacy of $(\pi(s), \pi(t))$ in R. □

6.3.2 Input Program Translation

We are now ready to describe the translation of program $\mathcal{P}$ from its components (1) through (4) (beginning of this section): The new program $\widehat{\mathcal{P}}$ consists of the same variable $\vec{v}$ with range V, further variables $\hat{d}_r$, $r \in [1..z]$ with range $[1..l]$ and variables $n_1, \ldots, n_l$ with range $[0..n]$. Every edge of the skeleton is translated into a statement as follows:



```
if $n_X > 0 \;\wedge\; \hat{g}$ then
    $update1(g)$
    $n_X := n_X - 1$                (6.4)
    $n_Y := n_Y + 1$
    $update2(\mathcal{A})$
```

The condition $n_X > 0$ ensures that there is a process in local state X. Guard g is translated into $\hat{g}$ by translating its constituent basic guards, as follows. Each basic guard on local states is replaced according to table 6.1. Expressions over $\vec{v}$ are unchanged. For an ID-sensitive variable d, guard $d = i$ is replaced by $\hat{d} = X$, guard $d \neq i$ by $\hat{d} \neq X \vee n_X \geq 2$. As an intuition for this last translation, if $n_X \geq 2$, there is a process i in local state X with $d \neq i$; guard $d \neq i$ is true for that process.

As we have seen in the example in section 6.2, in some situations updates of an ID-sensitive variable $\hat{d}$ are necessary merely because the local state of a process changes, although d is not assigned in the original program. (In fact, if it is assigned, the following updates can be skipped, as they will be overwritten.) Function *update1* performs these updates on $\hat{d}$:

g	$d = i$	$d \neq i$	otherwise

update1 (g)	$\hat{d} := Y$	*no-op*	**if** $\hat{d} = X$ **then**
			if $n_X = 1$ **then**
			$\hat{d} = Y$
			else
			$\hat{d} :\in \{X, Y\}$

The "otherwise" column includes the case that d does not even occur in the guard. This part of the translation looks cumbersome; we motivated it in the example in section 6.2.

Function *update2* implements updates of global variables that are due to action $\mathcal{A}$ itself. It leaves *no-op* and assignments to $\vec{v}$ unchanged. Assignments to an ID-sensitive variable d are translated as follows:

$\mathcal{A}$	$d := i$	$d :\in ([1..n] \setminus \{i\})$	$d :\in [1..n]$

update2 $(\mathcal{A})$	$\hat{d} := Y$	**if** $n_Y = 1$ **then**
		$\hat{d} :\in (\{L : n_L > 0\} \setminus \{Y\})$
		else
		$\hat{d} :\in \{L : n_L > 0\}$
			$\hat{d} :\in \{L : n_L > 0\}$

Applying the translations described to program $\mathcal{P}$, we obtain a program $\widehat{\mathcal{P}}$ whose execution semantics is inherited from that of $\mathcal{P}$: at every cycle, a statement is nondeterministically chosen and executed. (If the guard of the statement is not satisfied, its execution is equivalent to a *no-op*.) To show that $\mathcal{P}$ and $\widehat{\mathcal{P}}$ are in some sense equivalent, we compare their Kripke structures:

Definition 24 *Program $\widehat{\mathcal{P}}$ defines a Kripke structure $\widehat{M} = (\widehat{S}, \widehat{R}, \widehat{L})$ as follows: $\widehat{S} = V \times [1..l]^z \times [0..n]^l$, and $\widehat{R}$ contains all pairs $(\hat{s}, \hat{t})$ such that there is a (nondeterministic) statement in $\widehat{\mathcal{P}}$ and an execution of it that modifies $\hat{s}$ into $\hat{t}$. Finally,*

$\widehat{L}$ labels a state with fully symmetric expressions (over local state variables and the ID-insensitive global variables) that are true at any specific state represented by $\hat{s}$.

6.4 Verifying the Generic Program

In anticipation of using $\widehat{M}$ for verification, we first show that it is much smaller than M:

Theorem 25 *Let $\overline{M}$ be the quotient of the original program's structure M and $\widehat{M}$ the structure from definition 24. $\overline{M}$ is subgraph-isomorphic to $\widehat{M}$ via the mapping*

$$b \colon \overline{S} \to \widehat{S} : \quad b(\vec{x}, k_1, \ldots, k_z, s_1, \ldots, s_n) = (\vec{x}, s_{k_1}, \ldots, s_{k_z}, n_1, \ldots, n_l)\,.$$

with $n_L := |\{j \in [1..n] : s_j = L\}|$ for $1 \le L \le l$.

Proof (sketch): We have to show that $\overline{M}$ is isomorphic to a subgraph of $\widehat{M}$. This subgraph is obtained from $\widehat{M}$ by restricting the counter tuples $(n_1, \ldots, n_l)$ to the set $\widehat{S}^{\#}$ of tuples that add up to n, the number of processes[1]. Intuitively, this is an obvious invariant of the counter-abstracted program, since the sum of all counters equals the total number of processes. The induced subgraph $\widehat{M}^{\#}$ of $\widehat{M}$ therefore comprises $\widehat{M}$'s reachable part, independently of the initial state(s).

One now shows that b is a bijection between $\overline{S}$ and $\widehat{S}^{\#} \subseteq \widehat{S}$, which follows from completeness and uniqueness of the representatives chosen from the symmetry equivalence classes. The isomorphism property between $\overline{M}$ and $\widehat{M}^{\#}$ then follows since the unique orbit representatives in $\overline{S}$ and the counter tuples in $\widehat{S}^{\#}$ are just different notations for the same concept: a complete choice of states such that no two are identical up to permutations of processes. □

[1] See also the example given earlier in section 2.1.2 on 17.

Function b in theorem 25 maps every state to its unique generic representative. Isomorphism is of course a stronger property than bisimulation; one difference is that bisimulation can exist between structures of vastly different sizes and is therefore a concept valuable for abstraction.

Since M is bisimilar to $\overline{M}$ (standard symmetry reduction) and $\overline{M}$ is isomorphic to (the reachable part of) $\widehat{M}$ (theorem 25), it follows by transitivity that M is bisimilar to $\widehat{M}$. As a corollary to theorem 25, since $\overline{M}$ is smaller than M by a factor of about $n!$, so is $\widehat{M}$. This paves the way for efficiently verifying properties of the original program $\mathcal{P}$.

Consider a model checking problem of the form $M, s \models f$. Formula f is formally defined as a temporal logic property with atomic propositions that are fully symmetric expressions on local state variables and expressions on the ID-insensitive variables. In practice, a model checking algorithm that explores $\widehat{M}$ is easier to implement by translating f into a formula over counter expressions. That is, the fully symmetric expressions on local state variables are treated like basic guards and translated as in table 6.1.

We summarize with the remark that the translation of the program as well as of the formula can be done fully automatically using the rules established in this and the previous section. The complexity of the translation is roughly linear in the size of the program text or the formula, respectively.

6.5 BDD-Complexity of the Generic Program

In this section, we show how the statements of the generic program, obtained in section 6.3.2, can be encoded in a BDD. After all, the main motivation for this work is to circumvent the orbit problem for symbolic model checking. We remark, however, that the generic program is also useful with explicit-state model checking: the advantages mentioned near the end of section 6.1 apply to it, too; the situation is just not as desperate as it is for symbolic model checking, where the orbit relation complexity essentially annihilates the benefits of symmetry reduction when applied naively.

We also estimate the sizes of those BDDs as a function of the number of processes n, the number of local states l and the size of the input synchronization skeleton. In this section we ignore the existence of the ID-insensitive variable $\vec{v}$: since expressions involving it are subject to no restrictions, BDD sizes cannot be estimated. However, those expressions are unaffected by the translation; hence they do not contribute any change in BDD size. As usual, we use the prime notation, as in $\hat{d}'$, to indicate the next-state value of a BDD variable.

The generic structure $\widehat{M} = (\widehat{S}, \widehat{R}, \widehat{L})$ is the disjunction of statements of the form in (6.4) in section 6.3.2. BDDs that implement those statements can be obtained as follows:

if-then-else: this statement can be implemented using a common operation (**ITE**) for BDDs. The complexity is low-degree polynomial in the complexity of the condition and the **then** and the **else** part.

$n_X > 0$**:** it holds iff there is at least one *true* bit among the $\lceil \log(n+1) \rceil$ bits representing n_X. This can be implemented as a disjunction over all those bits. The resulting BDD size is linear in the number of participating bits: $\mathcal{O}(\log n)$.

$\hat{g}$**:** it is a propositional combination of basic generic guards. Guards from table 6.1 can be realized as above with a BDD that compares the constant bit-wise against the counter variable; size $\mathcal{O}(\log n)$. Basic generic guards involving the ID-sensitive variable have the form $\hat{d} = X$ or $\hat{d} \neq X \vee n_X \geq 2$, which can again be verified bit-wise; these BDDs thus have maximum size $\mathcal{O}(\log l \log n)$ ($\hat{d} \in [1..l]$, $n_X \in [0..n]$).

update1(g)**:** The expressions inside the **if-then-else** for the (most complex) "otherwise" case are comparisons against constants. The entire statement can thus be encoded in a BDD of size $\mathcal{O}((\log l)^\alpha \cdot (\log n)^\beta)$, for small constants α and β.

$n_L := n_L \pm 1$: since the right-hand side is not a constant, a bit-wise comparison is impossible. The increment can be implemented by searching (using existential quantification) for a bit position i at which n_L is 0, n'_L is 1, for all preceding bits n_L and n'_L are identical, and for all succeeding bits n_L is 1 and n'_L is 0. The worst-case BDD size over two variables of $\lceil \log(n+1) \rceil$ input bits is $2^{2\lceil \log(n+1) \rceil} = \mathcal{O}(n^2)$.

***update2*$(\mathcal{A})$:** assignment $\hat{d} :\in \{L : n_L > 0\}$ can be realized with a BDD for the expression $\bigvee_{L \in [1..l]} (n'_L > 0 \ \wedge \ \hat{d}' = L)$ of worst-case size $\mathcal{O}((\log n \log l)^l)$. The BDDs for all operations in *update2*$(\mathcal{A})$ then have about this worst-case size.

We see that all parts of the translation of an edge can be expressed with a BDD that is low-degree polynomial in n, although it can be exponential in l. The latter complexity is subsumed by the complexity of the overall BDD for the transition relation $\widehat{R}$, which is also worst-case exponential in l. The reason is that the synchronization skeleton can have about l^2 edges; $\widehat{R}$ is thus a disjunction of up to l^2 expressions composed of the above forms. A disjunction operation can double the size of the BDD, but this worst case tends to be rare in practice.

Let us investigate how the relative sizes of n and l influence the benefit of generic representatives. Because of the $n \log l$ input variables of the BDDs for traditional local state vector representations of states (n variables of range $[1..l]$), an upper bound to the BDD size of the transition relations R and $\overline{R}$ is roughly l^n. For the generic method, we have l variables of range $[0..n]$, resulting in a worst-case BDD size for $\widehat{R}$ of roughly n^l. It can thus be assumed that the generic method is most useful if n is larger than l. Asymptotically, this is the case if l is a constant and n is considered a parameter. This situation occurs frequently in practice, since for a given application the number of local states is often fixed. When we evaluate the generic and the other techniques experimentally in chapter 9, we present such an instance.

6.6 Conclusions and Bibliographic Notes

The use of generic representatives, as vectors of local state counters are called in the context of symmetry reduction, has proved elegant and efficient, but also demanding. They are suitable for symbolic model checking of symmetric systems, provided we have the full symmetry group at our disposal. Experimental evidence scrutinizing the usefulness of counter abstraction is deferred until chapter 9, at which time we will have introduced the tool platform for such experiments.

In this chapter we have presented a translation scheme for a somewhat limited input language—we have assumed the program is preprocessed into local state transitions. Converting a more flexible input language, such as a programming language, into local state transition form is easy in principle, but can lead to another layer of state explosion if not done with care. We call this layer *local state explosion*; avoiding it is the topic of the next chapter of this dissertation.

Generic representatives seem to prove useful outside the symbolic domain as well: we translated some of the fully symmetric example programs coming with the Mur φ explicit state verifier [DDHY92] into generic representatives. For some examples, we obtained savings in both time and space of several orders of magnitude over Mur φ's symmetry reduction algorithms (using unique or multiple representatives).

Finite counters have been used previously to abstractly represent states of systems with many processes. The counters are sometimes truncated and assume only values up to some small constant, like 1 or 2, to increase the degree of abstraction, in particular to build finite-state representations of unbounded systems. A. Pnueli, J. Xu and L. Zuck used truncated counters with values 0, 1 or 2 to approximate the number of processes in certain local states in reasoning about symmetric parameterized systems [PXZ02]. Similar examples can be found in the work by A. Emerson and J. Srinivasan [ES90] on synthesis of parameterized programs and by F. Pong and M. Dubois on cache protocol verification [PD95].

Emerson and Trefler were the first to suggest the use of counters in connection with symbolic representation using BDDs [ET99]; the term *generic representatives* is due to them. Chapter 6 of this dissertation can be seen as extending their work by generalizing the syntax of input programs and establishing their full symmetry, and by quantitatively analyzing the sizes of the resulting BDDs. These results first appeared in [EW03].

A. Donaldson and A. Miller recently extend the generic representatives approach to probabilistic systems in connection with the Prism model checker [DM06].

Chapter 7

Improving Counter Abstraction by Counting Less

Overview. The goal of this chapter is to prepare a system of many identical components whose behavior is given in a high-level programming language for counter abstraction. We show that the question *what to count* has a strong impact on the performance of counter abstraction, and how to make the decision wisely, with the assistance of inexpensive front-end analysis techniques.

We have seen in the previous chapter that the strength of counter abstraction as a symmetry reduction technique comes from its ability to lower the structure complexity from exponential in n (classical state explosion) to polynomial in n. We have also seen that this comes at a price: the structure size is worst-case exponential in the number of local states, l. We thus have to focus part of our attention to making sure that this quantity does not get out of control.[1] In the previous chapter we avoided this problem by making the number of local states part of the input: we

[1] Making sure that the number n of processes does not get out of control is the objective of *parameterized verification*, which we address in chapter 11.

assumed the program was given as a synchronization skeleton, which can be viewed as a preprocessed, abstract form of a program in which values of local process variables are compressed into local states, and local computation is compressed into local state changes.

In this chapter we face reality: we consider a program parameterized by the identity of the executing process, and show how to extract a synchronization skeleton from it. This abstraction step can be viewed as a front-end to counter abstraction. If done naively, however, the resulting synchronization skeleton can be huge, ruining the expected benefit of the translation into local state counters. We address this problem in this chapter and show how this explosion can be ameliorated in practice. We first describe the problem and then present two independent remedies.

We remind the reader that this chapter, like the previous one, is about fully symmetric systems. Thus, when we speak of symmetry, we always mean with respect to the full symmetry group.

7.1 The Local State Explosion Problem

High-level modeling languages allow users to specify the behavior of processes in terms of (assignments to) global and local variables. The concept of local states is implicit and must first be extracted from the program. This is, at least in theory, straightforward. A local state is given by a valuation of the local variables. Quantitatively, let m be the number of local variables declared in a program template, and let $V_1, \ldots, V_m$ be the ranges of those variables. It follows that there are $|V_1| \times \ldots \times |V_m|$ possible local states of each process. If we naively introduce one counter per local state in order to perform counter abstraction, we obtain a number of counters that is exponential in m and hence in the input program size. We call this phenomenon *local state explosion.* Even a moderately large number of local states poses a problem since the size of the counter-abstract model depends exponentially on this number.

There are two approaches to this problem. The first is an old trick in verification: if the conceivable state space is too large, generate it on the fly. Indeed, instead of building a generic quotient structure, we could explore the state space over the original local state variables and use generic representatives only as a means to detect equivalence of states. This way, we build counter tuples on the fly, and unreachable local states are never considered. This procedure is justified for its purpose, but is, like many on-the-fly reduction techniques, not obviously realizable with symbolic data structures such as BDDs. Those data structures require us to declare up front the state variables (counters, in this case), so that BDD variables can be allocated for them.

In the rest of this chapter we investigate a second approach to ameliorating local state explosion. We present two techniques to statically analyze the program text, before counter variables are declared and any model is built. The goal of the analysis is to detect situations in which keeping a counter to monitor a local state is unnecessary. Such a situation might arise because the local state is known to be unreachable (section 7.2), or because some variable values in a local state are unused in the program and hence irrelevant (section 7.3).

As a caveat, both techniques are generally imprecise; they perform estimates. For instance, a comparatively cheap static analysis of the program text cannot reveal the exact reachable local state space. It is important that the estimate be conservative: every local state that needs to be monitored must be reported by the techniques, possibly some more. As long as this condition is satisfied, the exactness of counter abstraction is guaranteed; reporting more local states than required is an efficiency concern.

7.2 Amelioration through Local Reachability Analysis

Suppose L is a local state (i.e. a valuation of local variables) that is unreachable, by any process. In the counter-abstracted program, the corresponding counter n_L is invariably zero. If the unreachability of L is known a priori, we do not have to introduce n_L as a variable in the abstract program.

Formally, we define the *local reachability problem* as follows. *Given a local state L and a system of concurrent processes, determine whether there is a reachable global state in which some process is in local state L.* In general, this problem is of course a model checking problem by itself, characterized by the formula $M, s \models \mathsf{EF}\,\exists i L_i$. However, in order to perform counter abstraction we do not need to know the exact set of reachable local states; any over-approximation suffices. The better we approximate the set of reachable local states, the fewer counters we have to introduce, resulting in increased efficiency.

The set of reachable local states can be approximated in several ways. One solution is to build a Kripke model of the given program template, instantiated with only a single process, say process 1. Conditions on local states or local variables of other processes are treated conservatively, i.e. they are replaced by *true* if under an even number of negations, and by *false* otherwise. For example, the condition $\exists i : A_i$ is replaced by $A_1 \vee \mathit{true}$ and hence by *true*, whereas $\forall i : B_i$ is replaced by $B_1 \wedge \mathit{true}$ and hence by B_1. Conditions on global variables are likewise replaced by *true* or *false*, depending on their polarity. Assignments to global variables are discarded.

From process 1's perspective, we can view local states of other processes and global variables as part of an unpredictable environment. The above abstraction results in a Kripke structure that over-approximates the behavior of process 1. Since this *local* structure can be expected to be exponentially smaller than the *global* structure of the concurrent composition of the processes, standard reachability analysis can be performed on it. The outcome is the set of local states reachable in the local structure, which is an over-estimate of the set of local states reachable globally.

7.3 Amelioration through Live Variable Analysis

In this section, we assume the program executed by the processes consists of sequential code with individual control points that demarcate atomic actions. In this case, the local state space of a process contains a program counter, indicating the statement to be executed next. Analyzing the program allows us to estimate the way data are manipulated. We can exploit this information by only keeping track of values of variables that can possibly be used in the future.

Definition 26 (e.g. [Muc97]) *A variable x is called* live at a control point *if there exists a path to a future moment at which the value of x is used, and x is not assigned along the path. Otherwise, x is* dead at the control point.[2]

Consider the following example. Each of n processes has two local Boolean variables, *nonempty* and *locked*, and a program counter $PC \in [1..9]$. There is a global variable $q \in [0..n]$, which counts waiting processes. Process i's program is shown in algorithm 7.

Algorithm 7 Program text for process i

```
1: $nonempty_i := (q > 0)$; $q := q + 1$
2: if $nonempty_i$ then
3:     $locked_i := true$ //lock process $i$
4:     wait until $\neg locked_i$ //wait for someone to unlock process $i$
5: //execute restricted code here
6: if $q = 1$ then
7:     $q := 0$; goto 0
8: for some $j : PC_j = 3$ do $locked_j := false$ //unlock some process waiting at '4:'
9: $q := q - 1$; goto 0
```

[2] Like with the temporal operator F, "future" includes the present, i.e. the current program line.

Variable *nonempty* is live only at program line 2. It is used only there, and it is not live before reaching 2, since it is assigned right before in line 1. The consequence is that we do not have to remember the value of *nonempty* at any program line other than 2. For instance, the two local states $(4, \mathit{false}, \mathit{false})$ and $(4, \mathit{true}, \mathit{false})$, written in the format $(PC, \mathit{nonempty}, \mathit{locked})$, do not have to be distinguished, since they differ only in the value of *nonempty*, which is dead in line 4. Notice that both local states are otherwise legitimate and in fact reachable, so the technique from section 7.2 overlooks this redundancy. A similar argument holds for variable *locked*. It turns out to be live only at line 4, and thus needs to be remembered only at that point of the program.

How much does this analysis help counter abstraction? The straightforward approach introduces a separate counter for each conceivable local state, of which there are $|[1..9]| \times |\{\mathit{false}, \mathit{true}\}|^2 = 36$. In contrast, following the above analysis, at all lines except 2 and 4, no local variable other than the PC matters. For line 2, we only record the value of *nonempty*, and for line 4, only *locked*. The table below lists the local states that the program needs to monitor, using again the format $(PC, \mathit{nonempty}, \mathit{locked})$ with '$-$' for irrelevant values:

$$\begin{array}{lll} (1,\ -,\ -) & (3,\ -,\ -) & (5,\ -,\ -) \\ (2,\ \mathit{false},\ -) & (4,\ -,\ \mathit{false}) & \vdots \\ (2,\ \mathit{true},\ -) & (4,\ -,\ \mathit{true}) & (9,\ -,\ -) \end{array}$$

We have thus reduced the number of local states to keep track of from 36 to 11.

The formal justification for ignoring dead variables is as follows. Assume each process has a program counter PC and m other local variables $v^1, \ldots, v^m$. The concurrent execution of the program $\mathcal{P}$ by the processes in an asynchronous fashion defines a Kripke structure $M = (S, R)$. Recall that a global state $s \in S$ is given by a valuation of all global variables, and by assigning a local state to each process.

Definition 27 *Consider the binary relation $\sim$ on the local state space of each process, defined as $(PC_x, x^1, \ldots, x^m) \sim (PC_y, y^1, \ldots, y^m)$ if*

1. *$PC_x = PC_y$, and*

2. *$x^i = y^i$ for each $i \in [1..m]$ such that variable v^i is live at line PC_x.*

Relation $\sim$ can be extended to a relation $\approx$ on the global state space S by defining $s \approx t$ if s and t agree on all global variables and for each process p, the local states s_p and t_p of p in s and t, respectively, satisfy $s_p \sim t_p$.

Theorem 28 *Relation $\approx$ is an equivalence relation on S. Moreover, the quotient structure $\overline{M}$ of M with respect to $\approx$ is bisimilar to M with the canonical bisimulation relation $B := \{(s, [s]) : s \in S\}$ relating a state to its equivalence class under $\approx$.*

Proof: see appendix A.2. □

Counter abstraction of the dead-variable reduced structure $\overline{M}$ can be implemented fully automatically, and without first building $\overline{M}$, as follows. Determining live variables is a *data flow analysis* problem. The result is, for each value of the program counter, a list of the variables that are live at the corresponding line. Stepping through the program, we create a counter variable for each *partial valuation* of the local variables of the form $(PC, x^1, \ldots, x^k)$ such that x^i is a value of local variable v^i, and v^i is live at the given PC. Dead variables are not expanded into possible local states.

7.4 Conclusions and Bibliographic Notes

We have shown how to ameliorate the most severe efficiency issue of counter abstraction, local state explosion. Both techniques presented in this chapter are performed efficiently on the source code of the program. The techniques are static, i.e. they do not require (partial) execution of the program and ignore communication between components. They can be understood as a fast front-end to counter abstraction. Experimental results are presented in chapter 9, where we—among others—compare counter abstraction with and without the techniques in this chapter.

A point to note is that live variable analysis (section 7.3) requires an input model with highly predictable flow of control, such as a sequential program, as opposed to, say, a set of rules among which the next to be executed is nondeterministically chosen. In contrast, local reachability analysis via the local Kripke structure (section 7.2) is fit for any input model.

Let us return to the observation that the overall benefit of counter abstraction depends on the ratio between the number of local states l and the number of participating processes n. Since the counter-abstracted system can be shown to have size $\mathcal{O}(n^l)$, compared with $\mathcal{O}(l^n)$ for the original system, the case $n \gg l$ promises most benefits. If $l \gg n$, then at any time during execution most counters are zero, by the pigeon-hole principle. This phenomenon is orthogonal to the possibility of local states being unreachable, or local variables being dead. A local state may be currently unoccupied by any process, but generally reachable and refer to live variables only. Conversely, we have argued that the technique based on live variable analysis manages to eliminate counters for local states that are reachable, so the counters are not always zero.

How can we address the possibility that at any time during model checking the counter-abstracted structure, many counters are zero? For explicit-state model checking we can use a compressed notation for the zero-valued counters. For symbolic model checking we can use *zero-suppressed* BDDs [Min01]. However, since the set of zero-valued counters varies over time, counters for all local states must still be declared initially. The advantage of the techniques presented in this chapter is that they reduce the number of counters before even building a symbolic model;

irrelevant ones are simply not present.

The results of this chapter were first published in [EW04]. They appear to be unique in their improvement of counter abstraction using static analysis. Recently, [FBG03], [WS02] and [Yor00] suggest static analysis techniques to optimize BDDs, albeit unrelated to counter abstraction. The potential savings come from choosing dummy values for dead variables, or from nondeterministic assignments to them. This might reduce the size of the BDD graph, but does not diminish the number of allocated BDD variables. In contrast, we observe that dead local variables typically result in many redundant (equivalent) local states. All but one of them can be eliminated, significantly reducing the number of counters. This is guaranteed to decrease the number of BDD variables and the size of the BDD graph.

In [BR00], the use of compiler optimization techniques similar to ours is suggested to reduce the number of BDD variables to represent reachable states, with different BDDs for different program points. In contrast, the goal of this chapter is to build a symbolic representation of the overall program, to enable symbolic model checking. This is possible since counter abstraction (which is of no concern in [BR00]) allows us to incorporate live variable information into the abstract state representation, by creating local state counters judiciously.

Dataflow analysis is a well-investigated static technique often used by compilers to increase program efficiency, such as by optimizing register allocation. An elegant formulation of this technique is given by flow equations; the data flow information then corresponds to fixpoints of these equations (see for example [Muc97]). The complexity of dataflow analysis in practice is usually low-degree polynomial in the size of the input program.

Chapter 8

DySyRe—Symbolic Verification of Symmetric Systems

Overview. In this chapter we present "DySyRe" (we prefer to pronounce it like the word "desire"), a symbolic CTL model checker and experimental platform for finite-state systems. We begin with a general description about the intended purpose of DySyRe, and then present its input language and the property language it understands. Appendix B contains an example protocol description in DySyRe.

We call the tool DySyRe since it was originally an implementation of *Dy*namic *Sy*mmetry *Re*duction. Today it incorporates many algorithmic ideas from part II of this dissertation, namely dynamic symmetry reduction (chapter 5) including the lifting of error traces (section 5.4), handling of ID-sensitive variables (section 5.5.2), and a front-end for counter-abstracting a symmetric program (chapter 6). The tool also includes algorithms for plain model checking (which can be used to verify counter-abstracted programs), as well as model checking under traditional symmetry reduction schemes such as using the orbit relation or multiple representatives (section 4.3.3).

DySyRe comes with an extensive C++ library for system modeling using BDDs. The library also contains a CTL model checking engine, featuring both future-time and past-time temporal operators. The tool uses the CUDD BDD package for the underlying decision diagram manipulations [Som].

8.1 Purpose, Scope and Features of DySyRe

DySyRe is a symbolic verifier. It offers symmetry reduction using (i) the orbit relation, (ii) multiple representatives, (iii) the dynamic approach from chapter 5, and (iv) (in limited form) the generic representatives approach from chapter 6. Verification without exploiting symmetry is also possible (and often more efficient than the orbit relation and multiple representatives approaches). The generic representatives technique can be thought of as a front-end to model checking. DySyRe offers reduction with respect to the full symmetry group, the group of rotations, or any product thereof. Other basic groups with a reasonably small set of generators can easily be incorporated.

DySyRe especially supports experimentation. This is achieved through

- late binding of parameters in the system description (such as the number of process components): they need only be bound to values at runtime; no recompilation is required for different parameter values. In comparison, tools such as Mur φ and NuSMV do not allow runtime parameters in the system description.

- providing a flexible library for transition relation construction, instead of a specialized input language (which, however, requires some familiarity with the language of the library, C++).

- computing the set of states represented by the input CTL formula. If the set is small, the states can be listed in a compact format, and their number (i.e. the cardinality of the set) can be computed. For example, during the model building process, the user can visualize the set of reachable states of a small instance of a parameterized structure and thus potentially gain confidence in the model. In contrast, most model checkers just return “yes” or “no” (and possibly a counter example) in response to a formula.

The state space of the system is given through variables of structured types: Boolean, finite ranges, enumerations, records and arrays. The library provided to facilitate the transition relation construction offers routines to access these variables, further Boolean and comparison operators as well as basic arithmetic. Both synchronous and asynchronous systems can be modeled. DySyRe normally evaluates CTL formulas by returning the corresponding set of states. An exception is the specialized temporal operator **INV**, which carries out invariant checks and, in case of failure, prints an error trace in terms of the original state variables. DySyRe also accepts past-time temporal operators; for instance, invariant checks using **INV** can be done in a forward (from initial states) or backward (from error states) fashion.

8.2 Input Language of DySyRe

We sketch the way programs are written in DySyRe. This chapter of the dissertation is no substitute for the documentation of the tool,[1] which is why we abstain from presenting a formal grammar; the goal is to provide the reader with a feeling of what is expressible.

A DySyRe source file consists of the following regions:

```
[ Parameters ]
[ Constants ]
[ Variabe Order ]
```

[1] At the time of this writing, the tool and its documentation are available for download from `http://www.cs.utexas.edu/~wahl/DySyRe`.

```
[ Global Variables ]
[ Process-local Variables ]
[ Atomic Propositions ]
  Code
```

All regions except `Code` are optional, but one of `Global Variables` and `Process-local Variables` must exist. The order of these regions must be as given above. Below we briefly describe the meaning of each region; we mention again that appendix B contains an example protocol description in DySyRe for reference.

Parameters. This region specifies parameters of the system description such as the number of processes or number of memory cells. These parameters will appear as mandatory command line arguments in the final executable.

Constants. These are integral values that must be computable at compile time. That is, the initialization of a constant may not refer to parameters or system variables, only to literals and previously defined constants. The expression defining a constant may use standard arithmetic operators.

Variable order. This specifies which BDD variable order to use. There are three choices: concatenated, interleaved, and dynamic. The first two are typically used variable orders and are explained in equation (5.5) on 78. In these orders, each current-state BDD variable is immediately followed by its corresponding next-state variable. The third choice turns on dynamic variable reordering of the underlying BDD package CUDD. The default is concatenated.

Global variables. These variables exist exactly once in the model (are not replicated). The declaration recognizes the types **bool**, **enum** (symbolic range), **range**

(integral range), **array** and **record**, with their obvious meanings. Using arrays and records, types can be nested.

Recall from section 5.5.2 (83) the notion of ID-sensitive global variables. These must be declared as such by the user (failure to do so cannot be automatically detected and may lead to incorrect verification results). This also happens in this section of the input program declaration, using a special keyword. An ID-sensitive variable is always of type $[1..n]$; thus, type specifications are not allowed for such variables. We must, however, specify which *process group* the ID-sensitive variable belongs to (see below paragraph **Process-local variables**). This allows us to model systems that are orthogonal products of groups of processes with interchangeable behavior within one group. An ID-sensitive variable tracks the identity of a process in its group.

Process-local variables. They are specified once and automatically replicated for each process of the process group they belong to. More precisely, there can be any number of process groups: a *process group* declares a set of processes that are interchangeable, according to some symmetry group. Processes from different process groups are not interchangeable. A process group can be either of the *clique* type (all processes in the group are pairwise interchangeable) or the *ring* type (processes in the group can be rotated, such as in the dining philosopher's example).

Summarizing, a process group is specified by its type (clique or ring), the number of processes in it (which need not be compile-time evaluable, it may refer to parameters), and the local variables of the processes in this group. For the latter, the same comments as for global variables apply.

Atomic propositions. They are only declared in this region of the input program; their definition is part of the subsequent Code region.

Code. The purpose of this region is to define the atomic propositions (below we collectively use P to denote them) and the transition relation R, by means of C++ code. The user must define a function named **R** and an analogous function with the name **P**; these functions must return the BDD that encodes R or P. Each BDD is a Boolean expression that stands for a set of states if it defines P, or a set of transitions if it defines R. Some of the functions and operators available from the library that comes with the tool are listed in figure 8.1. Note that some of these functions are only suitable when defining P, some only when defining R.

In order to allow the modeling of a transition relation, each variable defined in the previous regions exists twice internally: once for the current-state value, once for the next-state value. Transitions are specified as pairs of current-state and next-state values of variables. For example, a transition that is due to the assignment `y:=y-1` is modeled using the BDD returned by the function call `dec(y,_y)`. Here, `y` stands for the bit vector that stores the current-state value of variable y, which is to be decremented. The expression `_y` stands for the bit vector that stores the next-state value of the same variable, which is to receive the computed value.

8.3 Property Language of DySyRe

After the input model is read by DySyRe and compiled into a BDD, the user is prompted to enter a "property", i.e. an expression that evaluates to a set of states. Such expressions are specified in DySyRe in an enriched dialect of CTL that, among others, adds past-time and invariant temporal operators to the language. An expression is evaluated into a set of states. DySyRe then prints whether this set is equivalent to *false* or to *true* or none of the above. In the latter case, the user has the option to print, in compact form if possible, the states in the set. For example, using the forward search operator **FS**, one can compute the set of bad states that are reachable from any initial state by evaluating a formula of the

```
BDD Zero(); // empty set (of states or transitions)
BDD One (); // full  set (of states or transitions)

BDD !A;        // negation            (set complement)
BDD A & B;     // conjunction         (set intersection)
BDD A | B;     // disjunction         (set union)
BDD A.Eqv(B);  // equivalence         (set equality)
BDD A.Diff(B); // (A & !B) | (B & !A) (set symmetric difference)
BDD A.Exor(B); // exclusive or

BDD A.IfThen(B); // if A then B
BDD A.Ite(B,C);  // if A then B else C

bool A.zero(); bool A.empty(); // true iff A is empty (false)
bool A.one (); bool A.full (); // true iff A is full  (true)
bool A.subset(B);              // true iff A subset B

// in the following, y may be a variable or a constant
BDD greater(x, y); // x >  y
BDD equal  (x, y); // x == y
BDD less   (x, y); // x <  y
BDD atMost (x, y); // x <= y
BDD atLeast(x, y); // x >= y

BDD dec(x, y); // y := x - 1
BDD inc(x, y); // y := x + 1

BDD invariant(x); // x = x'

// in the following, p is a process, sbn is a process group
BDD invariant    (p, sbn); // p          in group sbn invariant
BDD invariant_but(p, sbn); // all but p in group sbn invariant
BDD invariant_sbn(sbn);    // all        in group sbn invariant
BDD invariant_all();       // all invariant (globals may change)
```

Figure 8.1: Some of DySyRe's functions for creating BDDs

form (**FS** `init`) `&` `bad`. If no bad states is reachable, the result is the empty set. Otherwise, one can inspect the reachable bad states. If the number of bits in the state space is small (for example when instantiating the parameters with small values), this output is easy to grasp and can aid debugging. The print format reflects the variable and process group layout of the system.

Understood expressions/operators include the following:

False, True: empty and full set of states

States: state space S. This differs from the BDD for **True** if some variables have a domain whose size is not a power of two. For example, if a **range** variable [0..2] is declared, two bits will be reserved. Since the tuple 11 is invalid (only 00, 01 and 10), it is not part of the state space and not printed by **States**.

`!` | `&` `==` `=>` **:** Boolean connectives

`P`**:** name of an atomic proposition. Returns the set of states labeled P. Typical examples include `init` and `bad`.

EX, AX, EF, AF, EG, AG, EU, AU: future-time temporal operators, with standard CTL meaning

EY, AY, EP: past-time temporal operators, with the meanings "existential previous time", "universal previous time", "existential past". For example, the formula **EP** `init` represents the states reachable from `init`: from these states, there exists a past along which eventually `init` is true. **EY** and **AY** are forward image operators; **EP** can be used for forward reachability.

FS, BS: forward and backward search operators, respectively. **FS** and **BS** compute the same result as **EP** and **EF**, respectively, but they use *frontier set optimization* and are occasionally more efficient.

alpha, orbit: map the argument set of states to the set of representative states, and to the set of symmetry equivalent states, respectively.

INV_FS, INV_BS: invariant checking using forward or backward search, respectively. The result is either a confirmation of the invariance property, or a failure statement, followed by a counter example trace.

R: set of transitions of the system

size: number of states represented by the argument set. This may take long if the set is large.

The operators **INV_FS**, **INV_BS**, **R** and **size** do not compute a set of states. These operators can be used only at the top-level, i.e. they may not appear in the scope of any other operators.

8.4 Conclusions and Bibliographic Notes

We sketched in this chapter the DySyRe symbolic model checker, a tool that incorporates many of the techniques discussed so far in this dissertation. The emphasis with DySyRe is breadth of applicability, rather than user convenience. This is the reason for choosing a library to support modeling systems, rather than offering yet another modeling language, or re-using one from another tool. This feature, on the other hand, increases user demands, as knowledge of the library programming language (C++) is required. Also, by offering a broad modeling language, we cannot force the user to write models that are covered by the techniques implemented in the tool. Instead, the validity of the input model must be verified by an external mechanism, or taken on faith. The latter option is unsatisfactory, but even well-established tools like Mur φ (see below) sometimes sacrifice completeness in favor of verification efficiency; with Mur φ this is the case for the (not performed) verification of symmetry in loop statements.

DySyRe currently does not support fairness constraints. Liveness properties can be checked in the form of communal progress, or under specific fair scheduling strategies built into the model.

Distinguished examples of explicit-state model checkers using symmetry include Mur φ [MD], Spin [Hol97] and Smc [SGE00]. Mur φ contains one of the first serious implementations of symmetry reduction. It offers unique and multiple representatives, but is limited to invariant properties under full symmetry. A set of processes running the same program is known in Mur φ as a *scalar set*; several orthogonal scalar sets can be specified in an input program. Mur φ avoids the problems with equivalence detection and the orbit problem by (i) focusing on full symmetry, and (ii) being explicit-state, rather than symbolic for which no efficient symmetry reduction algorithms were known at the time of Mur φ's inception. The reduction strategy in Spin [BDH00] builds upon the theory of scalar sets developed for Mur φ by N. Ip and D. Dill [ID96].

Smc uses a randomized algorithm to map states to representatives. Notable are the incorporation of various fairness constraints and its support for *state symmetry* [ES96]. Due to explicit state enumeration, these tools are mostly confined to systems with a manageable number of reachable states.

UPPAAL [HBL$^+$03] is a real-time model checker for the verification of invariants and (some) liveness properties. It appears that only part of the state space is represented symbolically. The only truly symbolic model checker that emphasizes symmetry reduction is SYMM [CEJS98]. Like DySyRe, it offers the full range of CTL properties, but uses a sub-optimal reduction algorithm based on multiple representatives.

Chapter 9

Experimental Evaluation

Overview. In this chapter we provide experimental results that evaluate the techniques we introduced in this part of the dissertation. For each technique, we emphasize two aspects: (i) comparison of the technique against model checking without the technique but otherwise identical, and (ii) comparison of the technique against other techniques developed in this dissertation. The purpose of (i) is to show how much efficiency can be gained (or lost, in some cases) as a result of employing a technique. The purpose of (ii) is to discriminate the techniques against each other, especially regarding the types of problems for which they are successful.

Most experiments were performed using DySyRe as described in chapter 8. In some cases we use the Mur φ model checker. We reemphasize that DySyRe builds upon the CUDD BDD package. We ran the examples on a 1400 MHz Intel$^{\mathrm{TM}}$ Pentium$^{\mathrm{TM}}$ M processor with 256MB main memory. In columns titled “Problem”, a number behind the problem name indicates the number of processes in this instance of the (parameterized) problem. The abbreviations n and l always stand for the number of processes and the number of local states per process, respectively.

We usually give results both for memory consumption and time. Instead of measuring memory in bytes, we present a more easily reproducible figure. For BDD-based model checking, this figure is the maximum number of BDD nodes allocated at any time during execution ("Number of BDD nodes"). This number represents the memory bottleneck of the verification run. Assuming the same parameters of the BDD package (especially variable order), this number is independent of the machine used. Computation time is measured in seconds, minutes or hours, abbreviated s, m or h.

9.1 Generic Representatives

In this section we compare counter abstraction using generic representatives (chapter 6) against "traditional" methods for symbolic symmetry reduction, namely using the orbit relation (called "unique specific representatives") and using multiple representatives. We tested two examples.

The first, introductory example is a contrived mutual exclusion scenario that allows us to show how counter abstraction scales for varying values for n and l. Each process can be in one of the local states $L^1, \dots, L^l$, where L^{l-1} and L^l take the rôles of the trying region and critical section, respectively. The process must go through L^1 to L^{l-1} in this order before proceeding into L^l. In addition, the transition into L^l is protected by a binary semaphore, which is released again upon the process' return to L^1:

Transition				Guard	Action
L^i	$\rightarrow$	L^{i+1}	for $1 \leq i \leq l-2$	*true*	*no-op*
L^{l-1}	$\rightarrow$	L^l		$!sem$	$sem := 1$
L^l	$\rightarrow$	L^1		*true*	$sem := 0$

The second example is a variant of the list-based queuing lock with an atomic compare-and-swap instruction presented in [MS91]. We show the original code for this example in appendix C. The algorithm consists of an *acquire* and a *release* operation for a lock with the property that a process waiting for the lock spins only on process-local variables, instead of spinning on a global variable (like a semaphore). According to the authors of [MS91], spins on global variables can cause memory detention and impact system performance.

For both problems, we experimented with unique, multiple and generic representatives. We chose the set of multiple representatives *Rep* as follows:

$$\begin{aligned} r \in \mathit{Rep} \;\Leftrightarrow\; \exists i : 1 \leq i \leq l : \;\; & \text{process 1 is in local state } L^i \text{, and} \\ & \text{local states } L^j \text{ with } j < i \text{ do not appear in } r. \end{aligned}$$

For instance, using the first example (mutual exclusion) with $l = 3$, the states (L^1, L^2, L^1), (L^1, L^3, L^1), (L^1, L^3, L^2) and (L^2, L^3, L^2) belong to the set *Rep*. They are, however, not unique per orbit, in which the superscripts have to be in order (assuming the total order $L^1 \leq \ldots \leq L^l$).

For the mutual exclusion example, we verified the standard safety property $\mathsf{AG}\,\neg(L^l_i \wedge L^l_j)$ for all i, j with $i \neq j$, which—in generic terms—is expressed as $\mathsf{AG}\, n_{L^l} < 2$. For the second example, we verified that no two processes can acquire the lock at the same time, and also that there is no deadlock in the system. The latter means that it is never the case that all processes are simultaneously spinning in one of the two busy-waits that are present in the operations. Such a situation would cause a deadlock since a process cannot free itself from a busy-wait: it can only be unlocked by another processes.

These properties were verified using the standard symbolic fixpoint characterization of $\mathsf{EF}\, \mathit{bad}$. The variables were ordered in an interleaved fashion; dynamic variable reordering was disabled. Table 9.1 shows how the space requirements and running times of the three methods of symmetry reduction compare. For the unique specific representatives approach, we also display what percentage of the running time was used to just compute the orbit relation.

	Choice of n, l		Unique specific representatives		Multiple specific representatives		(Unique) generic representatives	
	n	l	Number of BDD nodes	Time (% orbit rel.)	Number of BDD nodes	Time	Number of BDD nodes	Time
M	8	4	114,894	8s (97%)	2,211	0s	703	0s
U	6	5	2,152,710	2:17m (97%)	6,612	0s	690	0s
T	16	16	–	>15h (100%)	132,377	7s	4,876	0s
E	64	16	–	–	599,561	3:18m	6,972	0s
X	128	128	–	–	–	>15h	69,060	10s
	256	128	–	–	–	–	78,060	13s
M	3	28	113,188	2s (79%)	30,614	0s	8,209	0s
C	4	28	9,478,195	1:13h (95%)	75,604	0s	12,361	1s
S	8	28	–	>15h (100%)	272,080	15s	63,113	16s
-	16	28	–	–	2,417,477	1:24h	183,324	2:09s
L	20	28	–	–	–	>15h	325,325	15:06s
K	60	28	–	–	–	–	3,109,874	7:04h

Table 9.1: Unique, multiple and generic representatives

For multiple and generic representatives, it can be seen that there is still room to grow memory-wise, but not necessarily so for unique representatives. Indeed, the main motivation for research on alternatives to unique representatives was the impractical BDD size of the orbit relation.

Further, the unique representatives approach spends nearly all of its time on the orbit relation construction. The use of multiple representatives clearly reduces memory and time requirements. The generic representatives solution outperforms, by several orders of magnitude, the other two both in memory and time, and hence in the size of problems it can handle.

Counter abstraction with and without BDDs. The purpose of the following example is to demonstrate the potential of counter abstraction compared with the multiple representatives approach even without using BDDs. We consider a variant of the classical scenario of r readers and w writers that compete for access to some data (see appendix B). The problem consists of two fully symmetric subsystems, but the global system is asymmetric (due to the readers' privilege to simultaneously access the data). The first algorithm applied to this problem is symmetry reduction using multiple representatives. The double column "Counter abstraction explicit-state" lists the results of applying the MUR φ verifier to the counter-abstracted program, i.e. non-symbolically. The third algorithm combines counter abstraction and symbolic representation.

Choice of r, w		Multiple representatives		Counter abstraction explicit-state		Counter abstraction with BDDs	
r	w	Number of BDD nodes	Time	Number of rules fired	Time	Number of BDD nodes	Time
8	8	19,853	1s	11,835	1s	1,082	0s
10	10	41,333	6s	25,784	1s	1,082	0s
16	16	223,770	108s	140,471	1s	1,379	0s
30	30	2,494,219	1:29m	1,482,854	2s	1,379	0s
100	100	–	–	159,625,349	162s	1,973	0s
1000	1000	–	–	–	–	2,864	2s

Table 9.2: Multiple representatives and counter abstraction w/o and with BDDs

We see from the table that counter abstraction is—even in its non-symbolic form—more successful than the symbolic multiple representatives approach, which suffers from symmetry reduction overhead. It should be noted that this overhead largely stems from the computation of the a priori representative mapping, which is oblivious of the simplicity of the transition relation of this problem. As the results on the right show, counter abstraction was most effective when combined with BDD-based symbolic model checking. The readers-writers scenario is characterized by a small number of local states. In such simple cases, techniques based on local state counters can be successful without employing front-end analysis techniques as described in chapter 7.

9.2 Live Variable Analysis

In this section we show the effect of local state reduction techniques on counter-abstracted symmetric systems (chapter 7). We use again a variant of the queuing lock algorithm, the original specification of which is presented in full in appendix C. The input is a program in a C-like language, with a predictable control structure. Such a program is amenable to a static analysis that establishes logical connections between different program locations, such as live variable analysis. The results are presented in table 9.3.

	Symbolic counter abstraction . . .			
	without LSR		with LSR	
Choice of n	Number of BDD nodes	Time	Number of BDD nodes	Time
5	16,583	1s	3,022	0s
10	71,224	29s	9,366	1s
15	156,421	110s	17,070	2s
30	785,411	25:53m	68,332	19s
50	2,643,540	3:34h	207,370	146s
70	5,586,017	12:16h	454,360	10:18m

Table 9.3: Symbolic counter abstraction w/o and with local state reduction (LSR)

The table teaches the following lesson. Counters corresponding to local states that have no bearing on the program behavior should be explicitly excluded from the BDD model. The fact that some conceivable local states differ only by irrelevant (dead) variables is not taken care of automatically.

9.3 Dynamic Symmetry Reduction

In this section, we exhibit the benefits and shortcomings of dynamic symmetry reduction (chapter 5) compared with techniques presented earlier in this dissertation. In table 9.4, we contrast the dynamic technique to multiple representatives and

	Multiple representatives		Counter abstraction		Dynamic symmetry reduction	
Problem	Number of BDD Nodes	Time	Number of BDD Nodes	Time	Number of BDD Nodes	Time
MsLock 10	369,239	1:15m	9,366	1s	24,092	15s
MsLock 20	4,407,127	4:05h	34,170	5s	139,990	9:35m
MsLock 30	–	>28h	68,332	19s	375,649	1:23h
CCP 03	16,522,710	13:12h	1,988,991	7:55m	14,088	1s
CCP 05	–	>35h	4,001,573	1:49h	74,754	14s
CCP 10	–	–	–	>18h	1,075,206	26:35m
CCP 15	–	–	–	–	4,947,726	6:17h

Table 9.4: Multiple representatives, counter abstraction and dynamic reduction

counter abstraction. Orbit relation based methods are excluded, since they take too much time or space for these examples.

The “MsLock” example is again the queuing lock algorithm from [MS91]. We see that counter abstraction performs better on it than dynamic symmetry reduction. The reason is that the number of local states is very small, which is the ideal battleground for the counter technique. We strongly emphasize, however, that the above numbers were obtained with the optimizations through local state space reduction in place (compare table 9.3).

The example denoted CCP refers to a version of a cache coherence protocol suggested by S. German, see for example [LS]. This protocol, on the other hand, is characterized by a large number of local states, which is why counter abstraction performs much worse on it than the dynamic technique. Figure 9.1 shows the memory results (number of BDD nodes) of table 9.4, but graphically using a histogram. The MsLock example is on the left, CCP is on the right. The number of BDD nodes is shown on a logarithmic scale.

Table 9.5 presents examples to which counter abstraction cannot be applied. The reason is that here permutations act upon states by not only changing the order

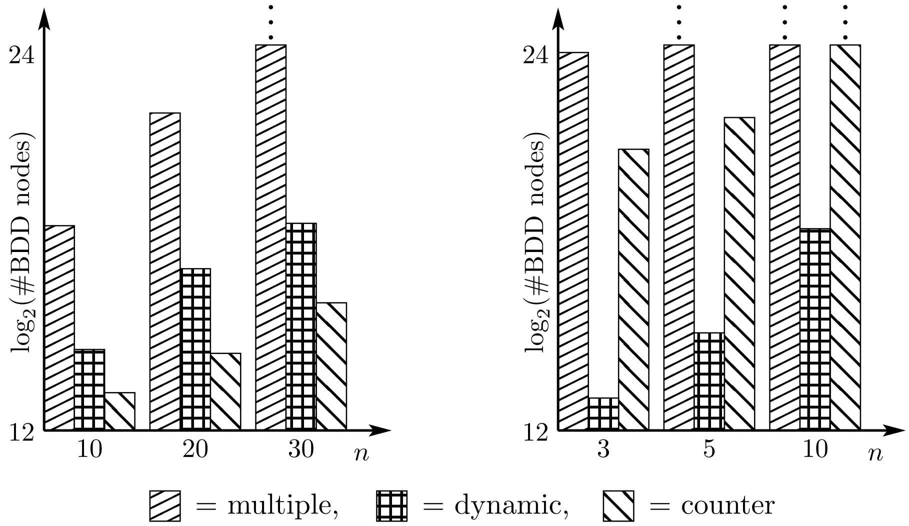


Figure 9.1: Multiple representatives, counter abstraction and dynamic reduction

of local states of components, but also their values. In this case, state equivalence cannot be detected using counting.

"C&S" (compare-and-swap) and "F&S" (fetch-and-store) are two extended versions of the queuing lock (provided in [MS91]). The "DL" example, taken from the Mur φ distribution [MD], is a distributed list protocol for processes in a FIFO queue sending and receiving messages and acting as a relay if asked to do so. Symmetry exists in both the processes and the messages. In this table we also show results of the verification run without symmetry reduction, where the intermediate BDDs quickly become huge. It is remarkable, however, how well plain model checking competes against the multiple representatives technique, which shows that the overhead incurred by the latter may not be worth the effort. The dynamic technique invariably outperforms the other two, for large problem instances by orders of magnitude.

	Without symmetry reduction		Multiple representatives		Dynamic symmetry reduction	
Problem	Number of BDD Nodes	Time	Number of BDD Nodes	Time	Number of BDD Nodes	Time
C&S 40	376,681	1m	157,470	25s	48,433	10s
C&S 50	–	>24h	4,259,627	37:34m	419,529	4:03m
C&S 60	–	–	–	>24h	6,246,717	2:10h
F&S 40	1,083,830	4:12m	413,036	2:02m	160,628	40s
F&S 50	–	>24h	–	>24h	2,017,634	29:43m
F&S 60	–	–	–	–	–	>24h
DL 30	861,158	28s	708,339	20s	60,394	2s
DL 40	6,380,209	4:35m	2,963,024	2:37m	213,448	5s
DL 50	–	>24h	13,580,042	29:30m	271,366	11s

Table 9.5: Plain model checking, multiple representatives and dynamic reduction

Part III

Extending the Scope of Symmetry Reduction

In this part of the dissertation we make symmetry reduction fit for more realistic scenarios. So far we have laid the foundations for exploiting symmetry in an ideal situation, where we are given a perfectly symmetric system of (a constant number of) n processes. This situation does occur in practice, and many protocols can be successfully verified or falsified using the techniques we have at our disposal so far. We have given examples near the end of chapter 9 of part II.

But often times we are not blessed with the ideal scenario, among others for the following two (very distinct) reasons:

1. the symmetry in the system may not be (known to be) exact, or
2. the number of components n may be variable.

Given the current pool of techniques, we have to use rather rough methods to attack these two problems. The first can be approached by over-approximating the system. That is, suppose the symmetry is inexact because not all transitions are preserved by all permutations. This means that not all local state changes can be executed by all processes. We can change the system by adding the missing behavior, to achieve a symmetric system. It is an over-approximation of the original, i.e. any ACTL* formula it satisfies is also satisfied by the original, but not vice versa, and we cannot say anything about non-ACTL* formulas without extra work. The disadvantages of this solution are obvious.

How to solve the second problem depends on how much we know about n. If its value is left completely open, we have to resort to general *parameterized model checking*. This is a very hard problem; we discuss it in chapter 11. In brief, the overall problem is undecidable, so we cannot always find a solution. If we know at least a range for n, such as $1 \leq n \leq N$, we can verify the problem for all n up to the bound N. This method is effective, but neither efficient nor elegant.

In this part of the dissertation we present better solutions to both of the above defects of symmetry. In chapter 10 we address the issue of unknown symmetry in the system. The method not only makes it unnecessary to detect the symmetry a priori (section 4.1 on 58). It can also deal with an amount of symmetry that cannot reasonably be expressed as a symmetry group. Instead, the solution we present *localizes* symmetry reduction to specific parts of the state space that may be unaffected by violations destroying the global symmetry, which render the principal methods from part II inapplicable.

In chapter 11 we take a look at the parameterized model checking problem and give a solution for the case that a bound to the parameter is known. This solution trades generality with respect to n in for generality with respect to system coverage. More precisely, the *aggregation* technique is applicable to arbitrary systems, whether symmetric or not, whether it adheres to other constraints of homogeneousness normally required by parameterized approaches or not. The price we have to pay is that such a solution necessarily does not apply to all size instances, but it does if an upper bound on the size is known.

Chapter 10

Reducing Partially Symmetric Systems

Overview. In this chapter we discuss the case of unknown but suspected approximate symmetry. We discuss what “approximate” means in this context and give an example. After defining some new vocabulary we introduce an algorithm for state space exploration on an approximately symmetric Kripke structure, examine implementation issues, and present experimental results.

All techniques for symmetry reduction we have looked at so far can be viewed as a sequence of two steps: (i) check that the system’s Kripke structure is symmetric with respect to some group (or find the largest such group), and (ii) symmetry-reduce the Kripke structure to a smaller one that can be more efficiently model-checked. The technique presented in this chapter does not require step (i) and incorporates step (ii) into the model checking process. If we don’t verify the system’s symmetry (be it directly on the Kripke structure or indirectly on the program text), then we cannot a priori build a quotient structure, so performing (ii) on the fly is our only option. Moreover, in contrast to dynamic symmetry reduction, we have to check each time we reach new states whether and how they can be collapsed due to equivalence to states seen before.

The new technique, which we call *adaptive symmetry reduction*, can be viewed as on-the-fly symmetry detection and reduction in one fell swoop. This appears more expensive than detecting symmetry up front on the program text, and it is. The advantages of delaying the detection to the model checking run are that (i) unreachable parts of the system's Kripke structure have no impact on the exploitable symmetry, and (ii) if the Kripke structure has symmetric substructures, those can be reduced to a subquotient. Standard symmetry reduction imposes an unreasonable punishment on systems where strict symmetry is violated merely because each process has, say, one transition that distinguishes it from other processes while all other behavior may be shared.

The adaptive approach to exploiting partial symmetry is to annotate each state, space-efficiently, with information about whether and how symmetry is violated along the path to it. More precisely, the annotation is a *partition* of the set of all component indices: if the path to the state contains a transition that distinguishes two components, their indices are put into different partition cells. Only components in the same cell can be permuted during future explorations from the state—the algorithm *adapts* to the state's history.

Suppose a given state can be reached along two paths: one with many asymmetric transitions and one with only symmetric ones. This state thus appears twice, once annotated with a fine partition, once with a coarse one. In order to analyze the state's future, we can assume that we reached it along the symmetric path and thus take full advantage of symmetry. The annotated state with the fine partition can be ignored; we say it is *subsumed* by the other one. Subsumption allows us to collapse many states during the exploration. The price we have to pay is that the adaptive algorithm, by its own means, is only suitable for reachability analysis.

Throwing away a state subsumed by another leads to an implicit reduced structure that is not bisimulation-equivalent to the original. This price is worth paying since it allows us to improve the analysis of systems with respect to safety properties, a significant and frequent type of formula.

We present adaptive symmetry reduction for the full symmetry group, $G = \mathit{Sym}[1..n]$, or orthogonal products of the full symmetry groups over subranges of $[1..n]$. For smaller groups, the symmetry reduction effect is correspondingly smaller to begin with. Partial, rather than exact, symmetry diminishes this effect further, which is why partial symmetry reduction for non-full symmetry groups is of limited interest. For these reasons, we omit the symmetry group G from the description in this chapter and always mean full symmetry.

10.1 What is Partial Symmetry?

A system is asymmetric if it is not symmetric. By definition 12, this means that the condition $\pi(R) = R$ is not satisfied for all permutations. When we use the intuitive term *partial symmetry*, we mean that for most transitions $r \in R$ and most permutations π, the condition $\pi(r) \in R$ holds.

Consider $r = (s, t) \in R$ and π such that $\pi(r) \notin R$. For an asynchronous system, $r \in R$ means that exactly one process, say i, changes its local state. In $\pi(r) = (\pi(s), \pi(t))$, this change applies to process $\pi(i)$. Since $\pi(r) \notin R$, process $\pi(i)$ is not allowed to perform this change, at least not in the context of the global state $\pi(s)$. Thus, we model partial symmetry in a system by the *asymmetry* of some local state transitions: those whose executability depends on (i) who requests to execute it (the process id), and (ii) what is the execution context (the current state). We formalize this way of describing a partially symmetric system in section 10.3.

Partitions. We introduce some terminology and notation that we use in this chapter of the dissertation. A *partition* of $[1..n]$ is a set of disjoint, nonempty subsets, called *cells*, that cover $[1..n]$. We use a notation of the form $|\,1,4\,|\,2,5\,|\,3,6\,|$ to represent the partition into the three cells $\{1,4\}$, $\{2,5\}$ and $\{3,6\}$. The coarsest partition $|\,1,\ldots,n\,|$ consists of a single cell, the finest partition $|\,1\,|\ldots|\,n\,|$ consists of n singleton cells. A partition $\mathbb{P}$ induces an equivalence relation on $[1..n]$: we write $i \equiv_{\mathbb{P}} j$ exactly if i and j belong to the same cell of $\mathbb{P}$.

We say a partition $\mathbb{P}$ of $[1..n]$ *generates* all permutations π on $[1..n]$ such that for all i, $i \equiv_{\mathbb{P}} \pi(i)$. These permutations form a group, denoted by $\langle \mathbb{P} \rangle$. For example, the partition $|\,1,4\,|\,2,5\,|\,3,6\,|$ generates a group of eight permutations. The coarsest partition $|\,1,\ldots,n\,|$ generates the entire symmetry group Sym_n . The finest partition $|\,1\,|\ldots|\,n\,|$ generates only the identity permutation.

10.2 Adaptive Symmetry Reduction—An Example

We introduce the idea of the technique presented in this chapter using an example. Consider the variant of the Readers-Writers problem shown in figure 10.1. There are two "reader" processes (indices 1, 2) and one "writer" (3). In order to access some data item, each process must enter its critical section, denoted by local state C. The edge from (the non-critical section) N to (the trying region) T is unrestricted, as is the one from C back to N. There are two edges from T to C. The first is executable whenever no process is currently in its critical section ($\forall j : s_j \neq C$, for

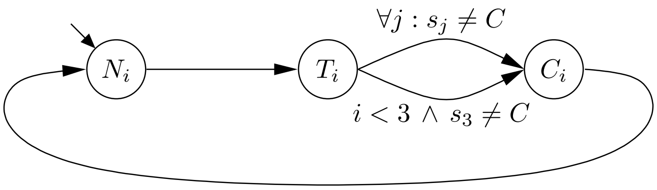


Figure 10.1: Local state transition diagram of process i for an asymmetric system

current state s). The second is available only to readers ($i < 3$), and the writer must be in a non-critical local state ($s_3 \neq C$). Intuitively, since readers only read, they may enter their critical section at the same time, as long as the writer is outside its own.

With each process starting out in local state N, the induced Kripke structure has 22 reachable states. The adaptive method presented in this chapter, however, constructs a reachability tree of only 9 *abstract* states (figure 10.2). An abstract state of the form XYZ represents the set of concrete states obtained by permuting the local state tuple (X, Y, Z). Consider, for example, the abstract state NNT, representing (N, N, T), (N, T, N) and (T, N, N). Guard $\forall j : s_j \neq C$ of the first edge from T to C is satisfied in all three states. Executing this edge leads to the successor states (N, N, C), (N, C, N), (C, N, N), succinctly written as NNC in figure 10.2.

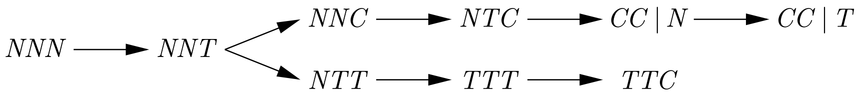


Figure 10.2: Abstract reachability tree for the model induced by figure 10.1

Now consider the abstract state NTC. None of the six concrete states it represents satisfies the condition $\forall j : s_j \neq C$. Thus, regarding steps from T to C, we have to look at the second—asymmetric—edge, guarded by $i < 3 \wedge s_3 \neq C$. Of the six represented states, two satisfy this condition with an index $i < 3$ such that $s_i = T$, namely (T, C, N) and (C, T, N). In both cases, the edge leads to state (C, C, N). We now have to make a note that this state is reached through an asymmetric edge. The edge's guard is invariant under the transposition (1 2), but not under any permutation displacing index 3. We express this succinctly in figure 10.2 as abstract state $CC \mid N$. Intuitively, permutations across the "|" are illegal; this abstract state hence represents neither (N, C, C) nor (C, N, C) (both happen to be unreachable in this system).

10.3 Representing Partially Symmetric Systems

As discussed at the beginning of this chapter, partial symmetry intuitively means that most local state transitions can be performed by most processes in most contexts. Thus, we expect a great deal of overlap in the behaviors of the processes. It is therefore economical to represent all processes' behavior by a common program and annotate the program with exceptions. As always, we describe the program abstractly using a local state transition diagram. That is, the system is specified as a number n of processes and a graph with local states as nodes. Local transitions, called *edges*, have the form

$$A \xrightarrow{\phi,\mathbb{Q}} B\,. \tag{10.1}$$

ϕ is a two-place predicate taking a state s and an index i. State s defines the *context* in which the edge is to be executed. The intended semantics is that $\phi(s,i)$ returns *true* exactly if in state s process i is allowed to transit from local state A to local state B. Predicate ϕ can be written in any efficiently decidable logic, such as propositional logic with basic arithmetic over state variables and index i. In figure 10.1 we have seen the predicate

$$\phi(s,i) \;\;=\;\; i < 3 \;\wedge\; s_3 \neq C\,. \tag{10.2}$$

It is asymmetric (and thus is the edge) since we can find s, i and a permutation π such that $\phi(s,i) \neq \phi(\pi(s),\pi(i))$. On the other hand, asymmetric edges are often symmetric with respect to a subgroup of $\mathit{Sym}[1..n]$. For instance, predicate (10.2) is invariant under the transposition $\sigma = (1\ 2)$, i.e. $\phi(s,i) = \phi(\sigma(s),\sigma(i))$ for all s, i. In common variants of the r-readers/$(n-r)$-writers problem, the asymmetric edges are immune to any products of permutations of $[1..r]$ and $[r{+}1..n]$. Such permutations are generated by the partition $\,|\,1..r\,|\,r{+}1..n\,|$.

Symbol $\mathbb{Q}$ in equation (10.1) stands for a partition generating the automorphism group of the edge, i.e. a set of permutations that preserve predicate ϕ. For the asymmetric edge in (10.2), we choose $\mathbb{Q} = |\,1,2\,|\,3\,|$. In approximately symmetric systems, $\mathbb{Q}$ is for most edges the coarsest partition, generating $\mathit{Sym}[1..n]$. For the remaining edges—those that destroy the symmetry—we expect the user to provide a suitable $\mathbb{Q}$. The high-level description of the edge often suggests a group of automorphisms; see section 10.7 for an example. If needed, a propositional SAT-solver can aid the verification of the automorphism property.

Letting l be the number of local states, an asynchronous semantics of the induced n-process concurrent system is given by the following Kripke structure: $S := [1..l]^n$, and R is the set of transitions $(s_1, \ldots, s_n) \rightarrow (t_1, \ldots, t_n)$ with the property that there is an index $i \in [1..n]$ such that

1. there exists an edge $s_i \xrightarrow{\phi,\mathbb{Q}} t_i$ with $\phi((s_1, \ldots, s_n), i) = \mathit{true}$, and
2. $\forall j : j \neq i : s_j = t_j$.

Note that $\mathbb{Q}$ is irrelevant for the definition of the Kripke structure. Extending the method to work with global variables is discussed in section 10.8.

Asymmetry in practice. Asymmetry is introduced into systems through restrictions on what process can perform what local state changes in what contexts, formalized by the edge condition $\phi(s,i)$. In practice, predicate ϕ may express conditions such as the following:

Informal predicate:	Formal predicate: $\phi(s,i) =$
"process 1"	$i = 1$
"all but process 1"	$i \neq 1$
"all processes whose right neighbor is busy"	$s_{i+1} = \mathit{busy}$
"all processes if all processes are free"	$\forall j : s_j = \mathit{free}$

The first two predicates often occur in systems with a hub process, such as a coordinator in a coherence protocol. Note that if $\phi(s,i)$ does not mention index i, the edge that ϕ appears on does not violate symmetry; the last predicate above is an example. Even if ϕ does mention i, full symmetry may be unaffected. In fact, we have seen in chapter 6 conditions on the predicates that guarantee this. If these conditions are violated for just one edge, the generic technique from that chapter cannot be applied. The point of the current chapter is to make the reduction process more flexible. Parts of the state graph that do not use any of the violating edges should be able to undergo full symmetry reduction.

10.4 Orbits and Subsumption

The goal of this chapter of the dissertation is an efficient exploration algorithm for the Kripke structure defined in the previous section. The algorithm accumulates states annotated with partitions that indicate how symmetry was violated in reaching this state. Thus, the formal search space of the exploration is the set $\widehat{S} := [1..l]^n \times Part_n$, where $Part_n$ is the set of all partitions of $[1..n]$. The partition is used to determine which permutations can be applied to the state in order to obtain the concrete states it represents. These permutations are those that do not permute elements across cells, i.e. those generated by the partition (see end of section 10.1):

Definition 29 *Let π be a permutation on $[1..n]$. For an n-tuple $s = (s_1, \ldots, s_n)$, let $\pi(s)$ denote the expression $(s_{\pi(1)}, \ldots, s_{\pi(n)})$, as usual. We extend π to operate on an element $\hat{s} = (s, \mathbb{P})$ of $\widehat{S}$ in the form*

$$\pi(s, \mathbb{P}) = \begin{cases} (\pi(s), \mathbb{P}) & \textit{if } \pi \in \langle \mathbb{P} \rangle \\ (s, \mathbb{P}) & \textit{otherwise.} \end{cases}$$

This mapping defines a bijection on $\widehat{S}$. Note that π never changes the partition associated with a state; if π is not generated by $\mathbb{P}$, it does not affect $(s, \mathbb{P})$ at all.

In standard symmetry reduction, algorithms operate on representative states of orbit equivalence classes. Systems with asymmetries require a generalized notion of an orbit that defines the relationship between states in $\widehat{S}$ and in S:

Definition 30 *The* orbit *of a state* $\hat{s} = (s, \mathbb{P}) \in \widehat{S}$ *is defined as*

$$Orbit(s, \mathbb{P}) = \{t \in S : \exists \pi \in \langle \mathbb{P} \rangle : \pi(s) = t\}\,.$$

We say that $\hat{s}$ represents t *if* $t \in Orbit(\hat{s})$*.*

Examples. For $n = 4$, consider the following states and the sizes of their orbits:

$\hat{s} = (s, \mathbb{P})$	orbit size
$(ABCD,\ \lvert 1,2,3,4 \rvert)$	$4! = 24$ (standard symmetry)
$(ABCD,\ \lvert 1,2 \,\vert\, 3,4 \rvert)$	$2 \times 2 = 4$
$(ABCD,\ \lvert 1,2 \,\vert\, 3 \,\vert\, 4 \rvert)$	$2 \times 1 \times 1 = 2$
$(ABCD,\ \lvert 1 \,\vert\, 2 \,\vert\, 3 \,\vert\, 4 \rvert)$	$1 \times 1 \times 1 \times 1 = 1$

If $\mathbb{P}$ is the coarsest partition $\lvert 1, \ldots, n \rvert$, then $Orbit(s, \mathbb{P})$ reduces to the equivalence class that s belongs to under the standard orbit relation.

Subsumption. Orbits in standard symmetry reduction are equivalence classes and as such either disjoint or equal. In contrast, the new orbit definition is not based on an equivalence relation. For example, the orbits of the four states in the table above form a strictly descending chain. It is unnecessary to remember all four states if encountered during exploration: the first subsumes the others:

Definition 31 *State* $\hat{s} \in \widehat{S}$ subsumes $\hat{t} \in \widehat{S}$*, written* $\hat{s} \triangleright \hat{t}$*, if* $Orbit(\hat{s}) \supseteq Orbit(\hat{t})$*.*

Examples. For $n = 3$, consider the following states and examples of what they subsume and don't subsume ($\mathbb{Q}$ is arbitrary):

$\hat{s} = (s, \mathbb{P})$	$\hat{s}$ subsumes:	$\hat{s}$ does not subsume:
$(ABC,\ \lvert\,1,2,3\,\rvert)$	$(ABC,\ \mathbb{Q})$, $(BCA,\ \mathbb{Q})$	$(ABB,\ \mathbb{Q})$
$(ABC,\ \lvert\,1,3\,\lvert\,2\,\rvert)$	$(ABC,\ \lvert\,1\,\lvert\,2\,\lvert\,3\,\rvert)$, $(CBA,\ \lvert\,1\,\lvert\,2\,\lvert\,3\,\rvert)$	$(BAC,\ \mathbb{Q})$
$(ABC,\ \lvert\,1\,\lvert\,2\,\lvert\,3\,\rvert)$	itself only	$(ABC,\ \lvert\,1,3\,\lvert\,2\,\rvert)$

Definition 31 provides no clue about how to efficiently detect subsumption. An alternative characterization is the following. Recall that $i \equiv_{\mathbb{P}} j$ iff i and j belong to the same cell within $\mathbb{P}$.

Theorem 32 *State $\hat{s} = (s, \mathbb{P})$ subsumes state $\hat{t} = (t, \mathbb{Q})$ exactly if*

1. *$i \equiv_{\mathbb{Q}} j \ \Rightarrow\ (i \equiv_{\mathbb{P}} j \vee t_i = t_j)$ is a tautology, and*
2. *$t \in Orbit(\hat{s})$, i.e. there exists $\sigma \in \langle\mathbb{P}\rangle$ such that $\sigma(s) = t$.*

Proof: see appendix A. □

Remark. Condition 1 is slightly weaker than the condition $i \equiv_{\mathbb{Q}} j \Rightarrow i \equiv_{\mathbb{P}} j$, which states that $\mathbb{P}$ is at least as coarse as $\mathbb{Q}$. As a hint why $t_i = t_j$ is needed for an equivalent characterization of subsumption, consider $\hat{s} = (AA, \lvert\,1\,\lvert\,2\,\rvert)$, which has a finer partition than $\hat{t} = (AA, \lvert\,1,2\,\rvert)$, but subsumes $\hat{t}$.

Condition 1 can, using appropriate data structures for partitions, be decided in $\mathcal{O}(n^2)$ time. In practice, violations are often detected much faster using heuristics such as comparing the cardinalities of $\mathbb{P}$ and $\mathbb{Q}$. Condition 2 requires checking whether $\mathbb{P}$ generates a permutation π that satisfies $\pi(s) = t$. This can be decided in $\mathcal{O}(n)$ time by treating each cell $P \in \mathbb{P}$ separately: we project both s and t to the positions in P and use a counting argument to verify that the projections are the same up to permutation.

Algebraic properties of subsumption. Relation $\triangleright$ is a *preorder*: it is reflexive and transitive. It is, however, neither symmetric (e.g. $(AB, \mid 1, 2 \mid) \triangleright (AB, \mid 1|2 \mid)$ but not vice versa) nor anti-symmetric (e.g. $(AB, \mid 1, 2 \mid)$ and $(BA, \mid 1, 2 \mid)$ subsume each other but differ). Thus, it is neither an equivalence nor a partial order.

We can derive an equivalence relation from a preorder by making it bidirectional: write $\hat{s} \bowtie \hat{t}$ iff $\hat{s} \triangleright \hat{t} \wedge \hat{t} \triangleright \hat{s}$. How is this equivalence related to the *orbit relation* on $\widehat{S}$, written $\hat{s} \equiv \hat{t}$ if there exists π such that $\pi(\hat{s}) = \hat{t}$?

Lemma 33 *For any $\hat{s}, \hat{t} \in \widehat{S}$, $\hat{s} \equiv \hat{t}$ implies $\hat{s} \bowtie \hat{t}$.*

Proof: Let as usual $\hat{s} = (s, \mathbb{P})$, $\hat{t} = (t, \mathbb{Q})$. Suppose $\pi(\hat{s}) = \hat{t}$ for some π. We show that $\hat{s} \triangleright \hat{t}$; the converse follows with a symmetric argument.

If $\pi \in \langle \mathbb{P} \rangle$, then $\pi(\hat{s}) = (\pi(s), \mathbb{P}) = (t, \mathbb{Q})$. Thus $\mathbb{P} = \mathbb{Q}$, which establishes condition 1 of theorem 32, and $\pi(s) = t$, which establishes condition 2. If $\pi \notin \langle \mathbb{P} \rangle$, then $\pi(\hat{s}) = (s, \mathbb{P}) = (t, \mathbb{Q})$, so $\hat{s} = \hat{t}$, which establishes $\hat{s} \triangleright \hat{t}$ by reflexivity of $\triangleright$. □

According to lemma 33, the orbit relation achieves less compression than subsumption: the latter is coarser, i.e. it relates more states. We note that in perfectly symmetric systems, where each state is (implicitly) annotated with the coarsest partition $\mid 1, \ldots, n \mid$, the three relations $\triangleright$, $\bowtie$ and $\equiv$ coincide.

10.5 State Space Exploration Under Partial Symmetry

We are now ready to present an algorithm for state space exploration on the (partially symmetric) structure $M = (S, R)$. The goal is to compute the set of states reachable under R from some initial state $s_0 \in S$. Technically, algorithm 8 below operates on elements of $\widehat{S}$; we later present a one-to-one correspondence between the states reachable in M and the states found by the algorithm.

In line 1, the initial state is annotated with the coarsest partition (indicating absence of symmetry violations so far) and put on the *Unexplored* and *Reached* lists.

Algorithm 8 State space exploration under partial symmetry

```
Input  initial state $s_0 \in S$
 1: $\mathit{Reached} := \mathit{Unexplored} := \{(s_0, |\, 1, \ldots, n \,|)\}$
 2: while $\mathit{Unexplored} \neq \emptyset$ do
 3:     let $\hat{s} = (s, \mathbb{P}) \in \mathit{Unexplored}$; remove $\hat{s}$ from $\mathit{Unexplored}$
 4:     for all edges $e = A \xrightarrow{\phi,\mathbb{Q}} B$ do
 5:         $\mathbb{R} := \mathit{glb}(\mathbb{P}, \mathbb{Q})$
 6:         $U := \mathit{unwind}(s, \mathbb{P}, \mathbb{Q})$
 7:         for all states $u \in U$ do
 8:             for all cells $R \in \mathbb{R}$ do
 9:                 if $\exists i \in R : u_i = A \;\wedge\; (u, i) \models \phi$ then
10:                     $v := (u_1, \ldots, u_{i-1}, B, u_{i+1}, \ldots, u_n)$
11:                     $\mathit{canonicalize}(v, \mathbb{R})$
12:                     $\mathit{update}(v, \mathbb{R})$
```

While available, one state $\hat{s}$ is selected from *Unexplored* for expansion.

Successors of $\hat{s}$ are found by iterating through all edges (line 4). We now have to reconcile two partitions: $\mathbb{P}$, expressing symmetry violations on the path to s, and $\mathbb{Q}$, expressing violations to be caused by e. Routine *glb* in line 5 determines the partition $\mathbb{R}$ such that $\langle \mathbb{R} \rangle = \langle \mathbb{P} \rangle \cap \langle \mathbb{Q} \rangle$. $\mathbb{R}$ can be computed as the greatest lower bound (meet) of $\mathbb{P}$ and $\mathbb{Q}$ in the complete lattice of partitions, which uses "at-most-as-coarse-as" as the partial order relation.

Edge predicate ϕ may not be invariant under permutations from $\langle \mathbb{P} \rangle$, but it is under permutations from $\langle \mathbb{Q} \rangle$ and thus from $\langle \mathbb{R} \rangle$. We account for this fact by unwinding s into a set of states to be annotated by $\mathbb{R}$ whose orbits exactly *cover* the orbit of $\hat{s} = (s, \mathbb{P})$, i.e. into a set $U \subseteq S$ that satisfies

$$\bigcup_{u \in U} \mathit{Orbit}(u, \mathbb{R}) = \mathit{Orbit}(s, \mathbb{P})\,. \tag{10.3}$$

The objective is of course to find a *small* set U with this property. In line 6, routine *unwind* returns the set $U = \{s\} \cup \{\pi(s) : \pi \in \langle \mathbb{P} \rangle \setminus \langle \mathbb{Q} \rangle\}$, which is easily seen to satisfy (10.3). This step can be a bottleneck; we discuss in section 10.6 how to avoid it in most cases and perform it as efficiently as possible in the remaining ones.

Processes with indices in different cells of $\mathbb{R}$ are distinguishable; we must consider these cells separately (line 8). Edge e can be executed if there is a process i in local state A such that (u, i) satisfies ϕ. If so, we let the process proceed, resulting in a new state v (line 10). In line 11, v is *canonicalized* within R: the sequence of local states with indices in R is lexicographically sorted.

The *update* function determines whether to add a new state $\hat{v}$ to the lists *Unexplored* and *Reached* (algorithm 9). If some state in *Reached* subsumes $\hat{v}$, nothing needs to be done; this also covers the case $\hat{v} \in Reached$. Otherwise (line 2), $\hat{v}$ is checked for errors (discussed below). Then, states that $\hat{v}$ subsumes are removed from *Unexplored*: such states are implicitly explored as part of $\hat{v}$ and are thus redundant. Finally, $\hat{v}$ is added to both lists.

Algorithm 9 Updating *Unexplored* and *Reached*: *update*$(v, \mathbb{R})$

```
Input  newly computed state $\hat{v} = (v, \mathbb{R})$
1: if no state in Reached subsumes $\hat{v}$ then
2:     check whether $\hat{v}$ represents a concrete error state
3:     remove from Unexplored each $\hat{w}$ such that $\hat{v} \rhd \hat{w}$
4:     add $\hat{v}$ to Unexplored and to Reached
```

States reachable from s_0 in M are related to states in *Reached* as follows.

Theorem 34 *Let $s_0 \in S$ and Reached as computed by algorithm 8. A state $s \in S$ is reachable from s_0 in M exactly if there exists $\hat{s} \in Reached$ that represents s.*

Proof (idea): by induction over the length of a path to a state reachable from s_0 in M. The complete proof is very technical and omitted here. □

Error conditions to be checked in line 2 of algorithm 9 need not be symmetric. For example, suppose the claim is that process 3 never enters local state X. Given $\hat{v} = (v, \mathbb{R})$, we determine the unique cell $R \in \mathbb{R}$ such that $3 \in R$. An error is reported exactly if the property $\exists i \in R : v_i = X$ evaluates to *true*.

If M has an error at distance d from s_0, then algorithm 8, if organized in a breadth-first fashion, detects it at distance d from the root of the abstract reachability tree. Using back-edges from each encountered node to its predecessor, an error path can be reconstructed and lifted to a shortest concrete path as usual.

Regarding line 3 of algorithm 9, the only reason not to remove $\hat{w}$ from *Reached* (but only from *Unexplored*) is to retain the ability to trace encountered errors back to the initial state, for which previously subsumed states may be needed. They are not needed for just finding errors or for termination detection.

10.6 Implementing the Exploration Algorithm

We discuss essential refinements of algorithm 8 and derive analytic results.

In approximately symmetric systems, most edges are symmetric, resulting in a search that annotates many states with the coarsest partition $|\, 1, \ldots, n \,|$. We encode this partition space-efficiently using the empty string. Further, a symmetric edge e in line 4 of algorithm 8 allows dramatic simplifications: Lines 5, 6 and 7 can be removed, as $\mathbb{R}$ equals $\mathbb{P}$ and U reduces to $\{s\}$. The test $(u, i) \models \phi$ can be factored out of the loop in line 8 (replacing i with 0), since it is independent of i (due to ϕ's symmetry). Almost the same simplifications apply if e is asymmetric but $\mathbb{Q}$ is coarser than $\mathbb{P}$ ($\langle \mathbb{Q} \rangle \supseteq \langle \mathbb{P} \rangle$), which is easy to test.

If $\mathbb{Q}$ is finer than $\mathbb{P}$, we must compute $U = \{s\} \cup \{\pi(s) : \pi \in \langle \mathbb{P} \rangle \setminus \langle \mathbb{Q} \rangle\}$. Doing this by enumerating $\langle \mathbb{P} \rangle \setminus \langle \mathbb{Q} \rangle$ is inefficient and unnecessary: state s likely contains redundancy in the form of duplicate local states (especially if there are more processes than local states). Thus, many permutations of $\langle \mathbb{P} \rangle \setminus \langle \mathbb{Q} \rangle$ result in the same state when applied to s. This redundancy can be avoided up front using *buckets*, i.e. sets of process counters for each local state, separately in each cell of $\mathbb{Q}$. Permutations outside $\langle \mathbb{Q} \rangle$ are applied to s by changing the contents of the buckets.

As a result, the complexity of *unwind* is proportional to $|U|$, which is usually much smaller than $|\langle \mathbb{P} \rangle \setminus \langle \mathbb{Q} \rangle|$. The set U itself is large only when $\mathbb{Q}$ is very fine, which is atypical for approximately symmetric systems.

To make the *update* function in algorithm 9 efficient, the list *Reached* is sorted such that states with local state vectors that are permutations of each other are adjacent, for example states of the forms $(AAB, \mathbb{P}_1)$, $(AAB, \mathbb{P}_2)$, $(BAA, \mathbb{P}_3)$. Given the newly reached $\hat{v} = (v, \mathbb{R})$, we first use binary search to identify the range in which to look for candidates for subsumption as the *contiguous* range of states in *Reached* whose local state vectors are permutations of v. The search in line 1 of algorithm 9 for states subsuming $\hat{v}$ can now be limited to this range.

We present complexity bounds for the adaptive exploration technique. Consider the abstract state space $\widehat{S} = S \times Part_n$, which is conceivably much bigger than S. The adaptive algorithm, however, only explores states not subsumed by others. Comparing the adaptive technique to standard symmetry reduction and to *plain* exploration ignorant of symmetry, our informal goal is to show that

$$complexity(adaptive) \quad \leq \quad complexity(standard) \quad < \quad complexity(plain)\,. \qquad (10.4)$$

If the automorphism group of the structure induced by a program is nontrivial, standard symmetry reduction is guaranteed to achieve some compression.[1] The meaning of "$\leq$" in (10.4) is that this compression is preserved by the adaptive technique.

To demonstrate this, we first quantify the effect of standard symmetry reduction on a program in the given input syntax. Call two processes *friends* if they are not distinguished by any edge, i.e. for each edge $A \xrightarrow{\phi, \mathbb{Q}} B$ there is a cell $Q \in \mathbb{Q}$ containing both processes. Friendship is an equivalence relation on $[1..n]$. Each class

[1] We overlook the pathological case in which only states of the form $(A, A, \ldots, A)$ are reachable.

of friends induces a group of permutations that can be extended to automorphisms of the program's Kripke structure. Friends, therefore, enjoy the following property:

Property 35 *Let F be a set of friends. Algorithm 8 reaches at most* $\begin{pmatrix} |F|+l-1 \\ |F| \end{pmatrix}$ *local state tuples over the indices in F.*

Proof (idea): The quantity in the property equals the number of representative states under standard symmetry reduction over the group $\mathit{Sym}\,F$ of all permutations of F and thus follows from property 15 (63) by extending $\mathit{Sym}\,F$ to act on the full range $[1..n]$. □

The orthogonal product of all groups of the form $\mathit{Sym}\,F$, for a set of friends F, is the largest symmetry group that can be derived from the program text. As a special case, if all n processes are friends, algorithm 8 reduces to standard symmetry reduction and introduces nearly no search overhead.

Whether the "$\leq$" in (10.4) is actually "$<$" or "$\ll$" depends on the way symmetry is violated and is hard to quantify in general. We observe, however, that for the adaptive technique, the notion of *friends* can be extended to include processes not distinguished by edges that are actually reached (executed) during the exploration. Unreachable asymmetric edges reduce the automorphism group, but have no effect on the adaptive algorithm. This observation is supported by experimental results.

10.7 Experimental Evaluation

We tested the adaptive method in a variety of experiments. We borrow a resource controller example from the work by P. Sistla and P. Godefroid [SG04, p. 729ff.]. In this example, process indices are partitioned into intervals of equal priority. In case of simultaneous requests, a server grants the resource to one of the highest-priority processes, thus introducing asymmetry. For a process belonging to the priority interval $[l_c..u_c]$, we annotate each asymmetric edge with the partition $|\,1,\ldots,l_c{-}1\,|\,l_c,\ldots,u_c\,|\,u_c{+}1,\ldots,n\,|$, separating higher, equal and lower priority.

In a first set of experiments, we compare the memory use of the adaptive technique with *plain* exploration oblivious of symmetry. Memory is measured by the (reproducible) number of reached states (memory in bytes is linear in this number, including the overhead due to the annotations). Figure 10.3 plots this number over various process counts n for the adaptive technique (small dots) and plain exploration (large circles) on a logarithmic scale. The graphs on the left and on the right differ in the priority scheme used. For $n = 18$, the plain algorithm reaches $1,310,716$ states on the left and $3,808,000$ on the right, whereas the adaptive algorithm reaches only 505 abstract states on the left and 316 on the right. The right scheme allows more compression due to larger priority classes; the 316 abstract states reached by the adaptive algorithm very compactly represent the $3,808,000$ concrete ones. In all cases, the adaptive algorithm took nearly zero time; for the plain algorithm the largest time measured is 7:16min.

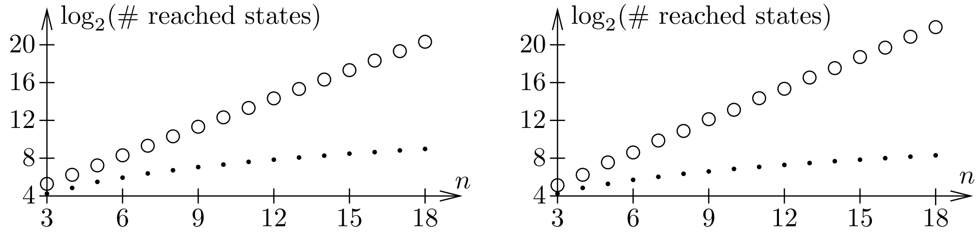


Figure 10.3: Comparing the adaptive technique (small dots) to plain exploration (large circles): reached states for $n/2$ small priority classes (left) and two large classes (right)

In a second set of experiments, we compare the memory use of the adaptive technique with standard symmetry reduction, based on the induced structure's automorphism group (figure 10.4). For the highly fragmented scheme on the left, the standard algorithm does quite poorly (thus again the logarithmic scale): for $n = 18$, it reaches $78,729$ states, compared with 505 adaptively. The maximum symmetry

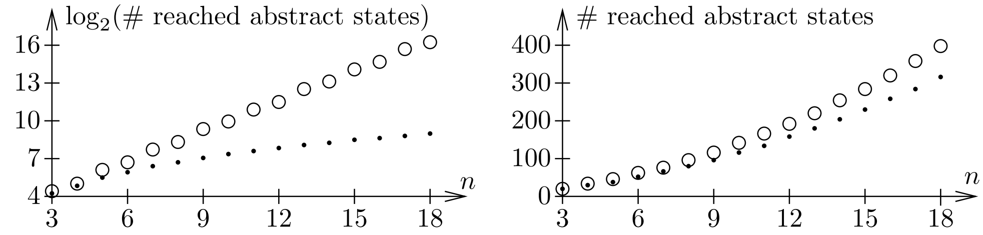


Figure 10.4: Comparing the adaptive technique (small dots) to standard symmetry reduction (large circles); priority schemes as in figure 10.3

group is the product of the nine transpositions (1 2) through (17 18), yielding a group size and expected compression factor of only $2^9 = 512$. This effect is much less severe for the less fragmented scheme on the right (linear scale), as is clearly revealed by the graph.

In a third set of experiments, we directly investigate how the adaptive method scales with increasing fragmentation; the idea for doing this is again borrowed from [SG04]. The resource controller example with k priority classes is run with a large number of 80 processes. The objective is to look for states where a process holds the resource while the resource is globally recorded to be free. In a first variant, denoted "$1, 1, \ldots, rest$", all priority classes but the last contain a single process; the last contains the rest. In a second variant, denoted "$2, 2, \ldots, rest$", all classes but the last contain two processes; the last contains the rest. We see from table 10.1 that the number of reached states grows roughly linearly with k; computation times are very reasonable. For fixed k, the fragmentation grows with increasing size of the initial k classes (1 vs. 2), since then the final class (hosting the majority of the processes) becomes smaller.

For $k \leq 5$, data obtained with the GQS-based method were provided in [SG04]. Those running times are an order of magnitude higher, although they of course depend on the machine used. Reproducible memory data for these examples (such as the number of reached states) were not given in [SG04].

k	n	"$1, 1, \ldots, rest$"		"$2, 2, \ldots, rest$"	
		Time	# states	Time	# states
2	80	1s	558	1s	789
3	80	2s	792	4s	1245
5	80	4s	1251	13s	2121
7	80	8s	1698	24s	2949
10	80	14s	2346	45s	4101
15	80	28s	3366	83s	5781
20	80	44s	4311	118s	7161
25	80	62s	5181	151s	8241

Table 10.1: Adaptive symmetry reduction against increasing fragmentation

10.8 Conclusions and Bibliographic Notes

In this chapter we presented a new adaptive method for exhaustive state space exploration. It is intended for, and efficient with, approximately fully symmetric systems, where many transitions are shared by most processes. Verification of this feature is not required; the method is exact for any input. We introduced the notion of subsumption: a state subsumes another if its orbit contains that of the other one. Subsumption induces a quotient structure with an identical set of reachable states. We focused on full symmetry, since this type is the most frequent and profitable in practice. The adaptive method can be implemented as well for rotation groups; critical is the ability to represent and manipulate groups succinctly. The implementation uses an explicit state representation. We believe the algorithm can be incorporated into the MUR φ model checker and extend its applicability to asymmetric systems.

In practice, system models often use global variables for communication. These may appear in the edge predicate ϕ, and they may be assigned as a side-effect of a transition. Most global variables have no bearing on the symmetry of the model. This includes synchronization variables such as semaphores, and the `busy` variable in the resource controller example [SG04], which indicates whether the server is currently serving a request. The definitions of *subsumes* and *represents* must be modified to ensure that these global variables are identical: $\hat{s} \triangleright \hat{t}$ requires agreement on them.

Conditions on and assignments to ID-sensitive variables (section 4.1, 54) may impact symmetry and must thus be taken into account when determining a suitable partition $\mathbb{Q}$ for an edge $A \xrightarrow{\phi,\mathbb{Q}} B$. Suppose, in a client-server model, after each service to a client, control must be passed back to the server, indicated by a pointer p reset to index 1 after each client transition. This pointer is an ID-sensitive global variable; the assignment $p := 1$ implies a partition $|\,1\,|\,2,\ldots,n\,|$ on the edge, or even finer, if symmetry is violated otherwise.

The results of this chapter first appeared in [Wah07]. Among the many related publications on the use of symmetry for state space exploration, one of the first to deal with partially symmetric systems is [ET99]. The authors present the notions of *near* and *rough* symmetry. In the former case, symmetry violations are allowed for transitions originating from symmetric states. Such transitions can serve as a tie-breaker in applications where priority decides which process gets to enter some exclusive local state first. The second notion is defined using an involved concept of coverage among transitions of individual components. Both near and rough symmetry are defined with respect to a Kripke structure; especially for rough symmetry it is unclear how to verify it on a high-level system description. Examples are limited to versions of the Readers-Writers problem.

This work was generalized in [EHT00] to *virtual symmetry*, the most general condition that allows a bisimilar symmetry quotient. A limitation of all preceding approaches is the existence of a strict precondition for their principal applicability. As with [ET99], it is left open whether virtual symmetry can be verified efficiently; the techniques presented in [EHT00] seem to incur a cost proportional to the size of the unreduced Kripke structure. On the other hand, bisimilarity makes these approaches suitable for full μ-calculus model checking, whereas the adaptive technique trades "property coverage" in for "system coverage".

We can compare the adaptive technique with virtual symmetry and its special cases in [ET99] using the example from section 10.2. The structure induced by the local state transition diagram in figure 10.1 on 141 is not *virtually* symmetric and hence not *nearly* or *roughly* so [EHT00, ET99]. To see this, consider the (valid) transition $(T, C, T) \rightarrow (C, C, T)$. Applying transposition (2 3) to it we obtain transition $(T, T, C) \rightarrow (C, T, C)$, which is invalid, but belongs to the structure's *symmetrization* [EHT00]. Virtual symmetry requires a way to permute the target state that makes the transition valid, which is impossible here. As a corollary, this structure is not bisimilar to its natural symmetry quotient.

Symmetry detection solves the problem of suspected but formally unknown symmetry by inferring structure automorphisms from the program text; a recent approach is given in [DM05]. This solution is principally different from ours. A structure automorphism is global in character, being defined over the transition relation. It ignores the possibility of a large part of the state space being unaffected by symmetry breaches. The adaptive approach, which can be viewed as *on-the-fly symmetry violation detection*, operates directly on the Kripke structure. As such, it can reduce local substructures with more symmetry than revealed by global automorphisms.

Closest in spirit to the present work is that by P. Sistla and P. Godefroid [SG04], who also target arbitrary systems and properties. A *guarded annotated quotient* is obtained from a symmetric super-structure by marking transitions that were added to achieve symmetry. As an advantage, this method can handle arbitrary CTL* properties. In the technique of this chapter, annotations apply to states, not edges, and seem more space-efficient; in [SG04] there can be multiple annotations to a quotient edge. Further, the adaptive method does not require any preprocessing of the program text, such as in order to determine a symmetric super-structure.

Chapter 11

Symmetry and Parameterized Reasoning

Overview. In this chapter we take a look at a much wider and more general problem in verification, known as *parameterized reasoning.* We discuss complexity issues and present an efficient solution to a decidable and practically relevant variant of this problem. This solution collapses many systems that belong to the same family into a single *aggregate* system, over which we can collectively verify properties of the family. The connection with symmetry reduction is given by the aggregate inheriting symmetry from the members of the family. We show an easy-to-implement way to exploit the symmetry in the aggregate.

In all techniques in this dissertation so far, we considered systems with replicated components and assumed to be given a number n of processes. That is, when presenting input to a model checker, the number n had to be instantiated to a constant. To increase generality, we can instead view n as a parameter, such that a value for n is no longer part of the input to the model checker. The *parameterized verification problem* is to decide whether a given temporal-logic property holds for all (i.e. infinitely many) instances of the size parameter. It is quite easy to see that this problem is related to the Halting problem for Turing machines in a way that makes it undecidable [AK86].

There are two principal ways of approaching parameterized verification algorithmically. One is to identify decidable subclasses of parameterized systems. To this end, many authors quite heavily restrict both the systems and the properties and give more or less efficiently verifiable conditions under which these properties hold for all instances. The other way is to realize that it is often possible and sufficient to consider a *bound* on the parameter size. Some applications suggest such a bound themselves, for example the number of components that fit on a particular circuit board. In other cases, verification engineers may find a correctness result that holds for a large number of components acceptable if all-inclusive parameterized techniques cannot handle their design.

In this chapter we develop a new approach to bounded parameterized verification. The goal is to verify—automatically and efficiently—temporal logic properties of an *arbitrary* parameterized system for a large finite range of values of the parameter. This can be accomplished, in an unsophisticated way, by analyzing the individual systems one by one, ignoring their common origin. This approach quickly becomes inefficient if the range for the parameter is nontrivial: in each run, both the modeling step and the verification are repeated, perhaps with only minor changes.

To address these shortcomings, we present a simple but effective technique to merge all instances in the given finite range into a single *aggregate* structure capable of simulating all systems from the range in one fell swoop. States of small systems (with few components) can be embedded in states of larger systems. The key is that we annotate each such embedding in a space-efficient way with the number of components in the embedded state, thereby making the merging lossless. Symbolic data structures such as BDDs can then be used to explore the aggregate structure in only little more time than (sometimes the same time as) it takes to traverse the largest of the original structures. This compares favorably with the cumulative time to analyze *all* structures one by one.

It is not obvious that the aggregate method outperforms the naive one. In fact, the findings of this method seem to contradict the principle of decomposing large systems into small, verifiable units, and then re-composing the results into a final report. The reason why in our case aggregation outperforms decomposition is that the components—here: instances of a parameterized system—are of similar form, suitable for a monolithic model. Moreover, we exert the power of symbolic data structures to compactly represent a large number of similar structures, at a cost much less than the sum of the costs to describe the individual entities.

The method developed in this chapter is applicable to an arbitrary, possibly *inhomogeneous*, finite system family, irrespective of any restrictions on the syntax of the system description or property. Given this much flexibility, it is well possible that the property under investigation is true for some but not for all instances, i.e. formulas may not be preserved across system sizes. In such cases, most traditional parameterized techniques are unlikely to be useful. In contrast, the technique presented in this chapter is capable of reporting the exact set of parameter values for which the property is incorrect, still with a single verification run. This provides an invaluable hint for debugging.

In the second part of this chapter, we build a bridge to symmetry reduction by considering the special case of families of symmetric systems. We show that the aggregate representation of all instances M_n by a single one, M, preserves the symmetry. Permutations, commonly used to formalize symmetry, are restricted to those that respect the special format of the states in the aggregate structure. We then demonstrate that with a careful encoding of M, this restriction can be ignored in an implementation: existing symmetry reduction algorithms can be applied without any changes. We emphasize that even though for homogeneous systems full parameterized verification *may* apply, a front-end is still required that checks whether the given system conforms to the imposed restrictions. Furthermore, this check may very well turn out negative, since symmetry alone is insufficient. None of this is of any concern with the aggregate method.

The aggregate approach can be viewed as a supplement to parameterized verification. It trades the benefit of solving the verification problem for infinitely many instances of a system, in exchange for greatly enhanced practicability. Indeed, the technique does not require any manual reasoning, imposes no restrictions on the input syntax, and is easy to implement.

11.1 Aggregating a Family of Systems

The goal of this chapter is to develop an approach to parameterized verification that works for any bounded family of systems derived from a synchronization skeleton parameterized by the number n of processes, and arbitrary CTL* properties. Let l be the number of local states occurring in the skeleton and AP be a set of atomic propositions to be used in temporal logic formulas. We omit global variables from the state description for now. Their presence is mostly immaterial to the techniques developed in this chapter, as we discuss in section 11.7. A global state s is thus a tuple $(s^1, \ldots, s^n)$ of local states of processes,[1] and we have $S_n = [0..(l-1)]^n$. Given two states s and t, let the notation $s^i \xrightarrow{g} t^i \in \mathit{SKEL}$ express that there is an edge in the skeleton from a node labeled s^i to a node labeled t^i such that s satisfies guard g (over local state variables). The transition relation R_n of the n-process concurrent

[1] Note that in this chapter, subscripts range over system instances and superscripts range over processes within one instance.

system is defined as usual as

$$R_n = \Big\{(s,t) : \exists i : i \leq n : \Big(s^i \xrightarrow{g} t^i \in \mathit{SKEL} \ \wedge \ \forall j : j \neq i : s^j = t^j\Big)\Big\}. \qquad (11.1)$$

With these definitions, the skeleton gives rise to a family $(M_n)_{n \in \mathbb{N}}$ of Kripke structures of the form $M_n = (S_n, R_n, L_n)$ with $L_n \colon S_n \to 2^{AP}$.

Let now N be an integer specifying the maximum number of processes we are interested in, i.e. we consider $n \leq N$. We represent all systems $M_1..M_N$ in a single *aggregate* structure by forming their disjoint union, in the following sense. A state of a particular instance M_n is given by the local states of n processes, which can be embedded in a local state vector of length N. In order to be able to recognize the state as a member of M_n, we fill the remaining $N - n$ vector positions with a fresh local state symbol, say \$. Every state vector is thus a sequence of non-\$ symbols followed by a sequence of \$ symbols. Intuitively, a process resides in local state \$ if its index is outside the range of the system to which the global state belongs.

Formally, we define a new Kripke structure $M = (S, R, L)$ over the state space $S = [0..l]^N$. Every state in S is a vector of length N over $l + 1$ local states. The embedding of the systems M_n in M is achieved as follows.

Definition 36 *For $n \leq N$, the* completion *of a state $s_n = (s^1, \ldots, s^n) \in S_n$ and of an edge $(s_n, t_n) \in R_n$, respectively, are defined as*

$$c(s^1, \ldots, s^n) = (s^1, \ldots, s^n, \underbrace{\$, \ldots, \$}_{N-n}) \in S, \quad c(s_n, t_n) = (c(s_n), c(t_n)) \in R. \qquad (11.2)$$

The completion of sets of states and sets of transitions is defined pointwise.

The completion upgrades states and transitions to members of the aggregate structure. We call a state $s \in S$ *proper* if there exists a number n such that s is of the form $(s^1, \ldots, s^n, \$, \ldots, \$)$, $s^j \neq \$$ for all $j \in [1..n]$. If s is proper, this number n is unique, called the *width* of proper state s. A state in S is proper of width n exactly if it is the completion of some state in S_n.

We are now ready to define the transition relation of the aggregate system:

$$R = \bigcup_{n \leq N} c(R_n). \tag{11.3}$$

R can be viewed as the disjoint union of the R_n, the disjointness being guaranteed by the fresh local state symbol \$. This definition ensures that the aggregate structure allows only proper paths, in the following sense.

Property 37 *For $(s, t) \in R$, both s and t are proper and have the same width.*

Proof: It is $c(R_n) = \{(c(s_n), c(t_n)) : (s_n, t_n) \in R_n\} \subseteq c(S_n) \times c(S_n)$. States in the completion of S_n are proper and have width n. □

Corollary 38 *All states along nonempty paths in the aggregate structure M are proper and have the same width.*

Finally, the labeling function L of M is defined as follows.

$$L(s^1, \ldots, s^N) = \begin{cases} L_n(s^1, \ldots, s^n) & \text{if } (s^1, \ldots, s^N) \text{ is proper of some width } n \\ \emptyset & \text{otherwise.} \end{cases} \tag{11.4}$$

We remark that L is well-defined since the width of a proper state is unique.

11.2 Efficiently Constructing the Aggregate System

In this section we illustrate how to efficiently implement the system representation outlined before with symbolic data structures such as BDDs. The main result is that building a BDD for the aggregate R differs only moderately from building a BDD for any R_n.

The first step is to make sure there is enough space to accommodate the additional $(l+1)$st local state, for each process. Representing state space S requires $\lceil \log(l+1) \rceil$ bits per process, which is equal to $\lceil \log l \rceil$ bits unless l happens to be a power of two. Hence, S can often be represented with no more bits than the largest of the original state spaces, S_N. When l is a power of two, the number of bits increases by one per process, compared with S_N.

Second, how do we implement the transition relation R? Equation (11.3) is suitable for proving theorems about the aggregate system, but not for implementing R, because it refers to the individual relations R_n, which we want to circumvent. Fortunately, there exists a different characterization of R, paving the way for a better solution.

Theorem 39 *Let the family of systems $(S_n, R_n)_{n \le N}$ be given as a synchronization skeleton. Then*

$$\bigcup_{n \le N} c(R_n) = \{(s,t) : \ \textit{s is proper of some width n, and} \\ \exists i : i \le n : \left(s^i \xrightarrow{g} t^i \in \mathit{SKEL} \ \wedge \ \forall j : j \ne i : s^j = t^j \right) \Big\} \tag{11.5}$$

(In the expression $s^i \xrightarrow{g} t^i \in \mathit{SKEL}$, guard g is evaluated over $(s^1, \ldots, s^n)$.)

Proof:

"$\Rightarrow$": Let $(s,t) \in c(R_n)$. Then by the definition of *completion*, s is proper of width n, and $((s^1, \ldots, s^n), (t^1, \ldots, t^n)) \in R_n$. By equation (11.1), there exists an index i with the property required in (11.5).

"$\Leftarrow$": Consider (s,t). From (11.1) and the second line in (11.5), we conclude $((s^1, \ldots, s^n), (t^1, \ldots, t^n)) \in R_n$. From the properness of s, we conclude $s^k = \$$ and hence $t^k = \$$ for $k > n$. Thus, $c(s^1, \ldots, s^n) = s$, similarly for t, and therefore $(s,t) \in c(R_n)$. □

This theorem provides the ingredients for an efficient implementation of R.

The left side of equation (11.5) is identical to the expression defining R on the right side of (11.3). The right side of (11.5) is almost identical to the right side of (11.1), which defines the transition relation R_n of a single system. The only difference is the requirement that s be proper. The reason for this requirement is that the width of a proper state tells us the number n of processes in the system instance that contains the state. This number is needed when a guard or an action of a skeleton edge refer to it. An example is a guard like $\forall i : s^i = T$, where n determines the range for i. Another example is the action $tok := (tok \pmod{n}) + 1$, where n determines the value at which the token is reset to one.

To implement R, we divide the skeleton edges in two classes: those whose guard does not refer to the system size n, such as a guard $\neg sem$ with a global semaphore variable sem, and those whose guard does refer to n, such as the guards in the paragraph above. For the former class, we simply translate every edge as if it was an edge of the largest system, M_N. For the latter class, we need an additional loop that iterates through the possible system sizes; see algorithm 10. In the figure, $e(p)$ stands for the propositional formula representing the size-independent skeleton edge e executed by process p. Similarly, $e(p, n)$ stands for the formula representing edge e executed by p in system M_n. The term $proper(n)$ in line 10 symbolizes the set of proper states of width n (expressed in current-state variables). It ensures that transition $e(p, n)$ can only be executed from a state that belongs to M_n. The computation of $proper(n)$ can be pulled out of the loop beginning in line 6.

We can see that for the second class of edges, the number of systems N we consider enters the complexity quadratically. We remark, however, that the majority of the edges in a skeleton defining a parameterized system usually belong to the first class, since dependence of transitions on the system size tends to destroy the regular system structure. Moreover, quite frequently edges that seem to depend on n can be rewritten such that the dependence goes away. Consider a conjunctive guard of

Algorithm 10 Implementation of the aggregate transition relation R

```
$R := \emptyset$
for $p := 1$ to $N$ do
    for all edges $e$ independent of the system size do
        $R := R \vee e(p)$
for $n := 1$ to $N$ do
    for $p := 1$ to $n$ do
        for all edges $e$ dependent on the system size do
            $R := R \vee (\mathit{proper}(n) \wedge e(p, n))$
```

the general form $\forall i : h(i)$. In the context of the aggregate structure, we can think of this guard as expressing the condition that every index i satisfy $h(i)$ *unless* i is greater than the width of the current state (i.e. i is “out of scope”). In this case the guard is to be ignored. Thus, the formula can be rewritten as $\forall i : (h(i) \vee s^i = \$)$ over the entire range $[1..N]$, independent of the actual system size. Similarly, disjunctive guards $\exists i : h(i)$ can be rewritten as $\exists i : h(i) \wedge s^i \neq \$$.

Finally, consider a system in which no edge depends on the system size. In this case, equation (11.5) can essentially be replaced by (11.1). In particular, the properness requirement need not be enforced in source or target states in R, since properness is propagated from the initial states during model checking (see next paragraph how proper initial states are constructed). In other words, it is then $R = R_N$, making the solution space-optimal. Although this exact situation may be rare in practice, it shows the asymptotic complexity of the technique as the number of dependencies on the system size decreases.

Implementing the labeling function L amounts to computing sets of states labeled with a particular atomic proposition. As an example, suppose I is a distinguished initial local state. For any n, this entails an initial global state of M_n with components $s^1 = \ldots = s^n = I$. According to equation (11.4), we can aggregate the initial states of all systems M_n into the following set of initial states of M:

$$\begin{array}{ll} 1. & (I, \$, \$, \ldots, \$) \\ 2. & (I, I, \$, \ldots, \$) \\ \vdots & \\ N. & (I, I, I, \ldots, I) \end{array}$$

A BDD for this set can efficiently be derived from the set P of proper states using the formula $P \wedge \forall i : i \leq N : (s^i = I \vee s^i = \$)$. The BDD representing the set of proper states of a certain width n has no more nodes than there are bits used to represent a state. It is computed with a loop over all conceivable indices $1, \ldots, N$. Indices greater than n are constrained to be equal to \$, all others are constrained to be different from \$. The set of all proper states (of any width) can be obtained as the union over sets of proper states of a specific width. These BDDs are all small in practice and have to be computed only once.

11.3 Verification over the Aggregate System

We are now ready to realize the main goal of this chapter of the dissertation: to reduce the verification of all systems up to size N to the verification of the aggregate system M. We accomplish this by establishing N bisimulations, one between each M_n and M, which contain pairs of a state and its completion:

Lemma 40 *For any $n \leq N$, the relation $s_n \in S_n \sim c(s_n) \in S$ is a bisimulation relation between structures M_n and M.*

Proof: Let $s_n = (s^1, \ldots, s^n) \in S_n$, hence $c(s_n) = (s^1, \ldots, s^n, \$, \ldots, \$) \in S$. (i) By the definition of the labeling function L, we have $L(c(s_n)) = L_n(s_n)$, since $c(s_n)$ is proper of width n. (ii) For t_n such that $(s_n, t_n) \in R_n$, we have $t_n \sim c(t_n)$. Since $(s_n, t_n) \in R_n$, we get $(c(s_n), c(t_n)) = c(s_n, t_n) \in c(R_n) \subseteq R$ by (11.3). (iii) Conversely, consider some $t \in S$ such that $(c(s_n), t) \in R$. By (11.3), there exists $m \leq N$ such that $(c(s_n), t) \in c(R_m)$. From $c(s_n) \in c(S_m)$, we derive $m = n$, hence $t \in c(S_n)$. This allows us to conclude the existence of t_n with $c(t_n) = t$, thus $(c(s_n), c(t_n)) \in c(R_n)$ and $(s_n, t_n) \in R_n$. □

We point out that there is in general no way to define a fixed initial state of M such that for every n, the initial states of M_n and M are bisimilar (if there was, the M_n would all be bisimilar to each other by transitivity). Instead, for each n an appropriate initial state of M must be chosen. This suffices for our purpose, which is to prove that a property true of all individual systems M_n is also true of the aggregate system M, and vice versa. For $n \leq N$, let $s_n \in S_n$ be the state of M_n with respect to which the property is to hold, and define

$$\Sigma = \{c(s_n) \in S : n \leq N\}. \tag{11.6}$$

All states $c(s_n)$ are proper and thus suitable as a start state of a path in M. We can now formulate the main result of this section:

Theorem 41 *Let f be a CTL* formula, and s_n, Σ as above. Then*

$$\forall n : n \leq N : M_n, s_n \models f \quad \textit{iff} \quad \forall s : s \in \Sigma : M, s \models f. \tag{11.7}$$

Proof: We exploit that structures with a bisimulation relation between them satisfy the same CTL* formulas with respect to bisimilar states (theorem 10).

$\Rightarrow$: Given $s \in \Sigma$, let s_n such that $s = c(s_n)$. Then $s_n \sim s$. Further $M_n, s_n \models f$ as given, and hence $M, s \models f$ follows with lemma 40.

$\Leftarrow$: Given $n \leq N$, we have $M, s \models f$ for $s = c(s_n) \in \Sigma$. Since $s_n \sim c(s_n)$, the claim $M_n, s_n \models f$ follows with lemma 40. □

Theorem 41 can be viewed as identifying a claim of the form "for all numbers n: ..." and a claim of the form "for all states s: ...". The latter is suitable to be approached with symbolic data structures that reason over sets of states, such as BDDs. Indeed, if BDD_f denotes the set of states of M that satisfy formula f, then the condition on the right of equation (11.7) is equivalent to $\Sigma \subseteq \mathit{BDD}_f$.

We remark that the meaning of formula f implicitly depends on n, namely through the labeling functions L_n. These may assign a given atomic proposition to different states in different systems; thus $\mathsf{EF}\, q$ may mean different things depending on the system.

How do negative verification results over M relate to the family of structures $(M_n)_{n \leq N}$? Assume the proof of $\forall s : s \in \Sigma : M, s \models f$ (right side of (11.7)) fails. Then there exists a nonempty set $V \subseteq \Sigma$ of states s such that $M, s \not\models f$. By the definition of Σ, all states in V are proper; the set $\mathit{width}(V) = \{\mathit{width}(s) : s \in V\}$ contains precisely the parameter values pointing to the delinquent systems. This set can give valuable information for debugging; section 11.6 presents an example of this phenomenon. Moreover, consider a particular $n \in \mathit{width}(V)$. If the failed verification of f over M admits a counterexample path, say p, then p can be mapped to a path in M_n by projecting every state along p to the first n components. The result is a valid counterexample path in M_n, due to the bisimulation between the structures: the two paths correspond.

Another consequence of the path correspondence is that the diameter and the girth of Kripke structure M, i.e. the distance between its most distant nodes and the length of its longest simple path, respectively, are equal to the maximum diameter, resp. girth, of any of the M_n. These numbers are important complexity measures in symbolic model checking. For example, the diameter is an upper bound on the number of image computations it takes for reachability analysis to converge. As a result, the time complexity of model checking the CTL formula $\mathsf{EF}\, \mathit{bad}$ over M, measured in number of image steps, is equal to the maximum time complexity, over all structures M_n, of model checking this formula over M_n.

11.4 Symmetric Families

Intuitively, due to the strong correspondence between the given system family $(M_n)_{n \leq N}$ and the aggregate M, one might expect that symmetry uniformly present in all of the M_n carries over to M. In proving this conjecture, one encounters the difficulty that the M_n have different numbers of replicated components. Thus permutations act on different sets of indices and cannot be compared across the M_n or related to M. A unifying solution is to let permutations from $\mathit{Sym}[1..N]$ act on all states, even with less than N components, after upgrading the states to dimension N using the completion operator. This step introduces the \$ symbol into the state, which, due to its special meaning, requires special treatment: we have to make sure permutations preserve the properness of a state. Otherwise, a transition between proper states could be permuted into a pair of improper states (by definition not a transition). We therefore first define a restricted permutation action, as follows.

Definition 42 *For any $\pi \in \mathit{Sym}[1..N]$ and $s \in S$, define*

$$\pi[s] = \begin{cases} \pi(s) & \begin{array}{l} \textit{if } s \textit{ is proper of some width } n \\ \textit{and } \forall i : i > n : \pi(i) = i \end{array} \\ s & \textit{otherwise,} \end{cases} \tag{11.8}$$

where as usual $\pi(s) = \pi(s^1, \ldots, s^N) = (s^{\pi(1)}, \ldots, s^{\pi(N)})$. This definition extends in the pointwise fashion to transitions and to sets of states and transitions. It can be shown that the relation $s \equiv t$ iff $\exists \pi : \pi[s] = t$ is an equivalence. The condition $\forall i : i > n : \pi(i) = i$ guarantees that no value i is permuted across the boundary between n and $n + 1$. Since $s^i = \$$ for all $i > n$ in a proper state s, it is irrelevant how permutations act on such i, as long as they respect this boundary. The weaker condition $\forall i : i > n : \pi(i) > n$ has the same effect. Regarding the "otherwise" case of equation (11.8), note that it applies not only to improper states, but also to proper states for which π violates the boundary.

Property 43 *For any $\pi \in Sym[1..N]$ and $s \in S$, s is proper if and only if $\pi[s]$ is proper. If both proper, they have the same width.*

Proof: If s is improper, then $\pi[s] = s$, so $\pi[s]$ is also improper. If s is proper, but π violates the properness boundary, then again $\pi[s] = s$, so $\pi[s]$ is proper. Otherwise, with n as in (11.8), $\pi(i) = i > n$ for all $i > n$, hence $s^{\pi(i)} = \$$. Due to bijectivity of π, we have $\pi(i) \leq n$ for all $i \leq n$, hence $s^{\pi(i)} \neq \$$, so $\pi[s]$ is proper; the claim of property 43 about the same width is immediate. □

We now define the notion of uniform symmetry for a parameterized system. In order to overcome the technical barrier that permutations acting on different systems have different domains, we use once again completions.

Definition 44 *The family $(M_n)_{n \leq N}$ of systems is called* uniformly symmetric *if*

$$\forall n : n \leq N : \forall \pi : \pi \in Sym[1..N] : \ \pi[c(R_n)] = c(R_n). \tag{11.9}$$

It is easy to see that $(M_n)_{n \leq N}$ is uniformly symmetric exactly if each system M_n satisfies $\pi(R_n) = R_n$ for all permutations on $[1..n]$. Definition 44 provides a closed formulation of this fact and refers to only a single permutation group, $Sym[1..N]$. This makes reasoning about uniformly symmetric systems convenient. We point out that in equation (11.9), permutations $\pi[\cdot]$ act according to equation (11.8), whereas in the expression $\pi(R_n) = R_n$, they act in the standard fashion; there is no notion of proper states in individual systems.

The main result in this section relates symmetry in the M_n and in M:

Theorem 45 *If $(M_n)_{n \le N}$ is uniformly symmetric, then M is fully symmetric.*

Proof: Let an arbitrary $\pi \in \mathit{Sym}[1..N]$ be given; we show $\pi[R] = R$:

$$\pi[R] \overset{(11.3)}{=} \pi \left[\bigcup_{n \le N} c(R_n) \right] \overset{(*)}{=} \bigcup_{n \le N} \pi[c(R_n)] \overset{(11.9)}{=} \bigcup_{n \le N} c(R_n) \overset{(11.3)}{=} R,$$

where $(*)$ follows from function application distributing over finite set union. □

Using this result, it remains to show that the quotient of M with respect to the orbit equivalence relation $\equiv$ and the special permutation action from equation (11.8) is bisimulation equivalent to M, so that we can verify CTL* properties over the quotient without losing information. This proof is similar to the argument used in standard symmetry reduction, provides no new insights and is thus omitted here.

11.5 Reducing Symmetric Families

Looking at the ungainly equation (11.8) defining permutation action, one might suspect that exploiting the symmetry in the aggregate system is more difficult or less efficient since only certain permutations can be effectively applied to a state. In the rest of this section, we show that this is not the case: restricting permutations in this way preserves the quotient size.

Symmetry reduction algorithms proceed by mapping an encountered state s to a unique representative of its equivalence class $\mathit{Orbit}(s)$ with respect to the orbit relation. As discussed in section 4.3.2, a common choice for the representative is the orbit's lexicographically least element, $\min_{\text{lex}}(\mathit{Orbit}(s))$, given some total order $\le_L$ on the local states. For example, in a three-process system with local states A and B, the global states (A, A, B), (A, B, A) and (B, A, A) form an orbit, which can be represented by the lexicographically least of the three states, (A, A, B).

We demonstrate in the following that such representatives can be computed without worrying about the special permutation action introduced in equation (11.8). Instead, permutations can be applied in the traditional way, with the same result:

Theorem 46 *Let s be a proper state. Then*

$$\min_{\text{lex}}\{\pi[s] : \pi \in \mathit{Sym}[1..N]\} \;=\; \min_{\text{lex}}\{\pi(s) : \pi \in \mathit{Sym}[1..N]\}. \tag{11.10}$$

Proof: Let n be the width of s, and let $P_{[s]}$ and $P_{(s)}$ be the two argument sets of the $\min_{\text{lex}}$ operator in equation (11.10). We first show $P_{[s]} \subseteq P_{(s)}$: Consider an element $\pi[s]$. If $\forall i : i > n : \pi(i) = i$, then $\pi[s] = \pi(s) \in P_{(s)}$. If not, then $\pi[s] = s = \mathit{id}(s) \in P_{(s)}$, for the identity permutation $\mathit{id} \in \mathit{Sym}[1..N]$. From this subset property we conclude $\min_{\text{lex}} P_{[s]} \geq \min_{\text{lex}} P_{(s)}$.

For the converse, let $s = (s^1, \ldots, s^n, \$, \ldots, \$)$. Since, by the choice of the numerical value of the special local state \$, $s^i \leq_L \$$ for all i, the state $\min_{\text{lex}} P_{(s)}$ has the form $m = (m^1, \ldots, m^n, \$, \ldots, \$)$. We have to show that $m \in P_{[s]}$, from which then $\min_{\text{lex}} P_{[s]} \leq m = \min_{\text{lex}} P_{(s)}$ follows. To map the *proper* state s to m, we can choose a permutation π that leaves all i with $i > n$ invariant ($\forall i : i > n : \pi(i) = i$) and permutes the first n components of s into their lexicographically least arrangement. For this permutation, $m = \pi(s) = \pi[s] \in P_{[s]}$. □

Theorem 46 shows that in order to map a proper state s to its orbit representative, there is no need to worry about the special permutation action. The key is, of course, that the local state of out-of-bounds processes, represented by \$, was chosen greater, with respect to the local state order $\leq_L$, than any other local state. Thus, representative mappings never move this symbol to the left in the local state vector and therefore preserve properness of states. As a result, the quotient of M with respect to the restricted permutation action defined in equation (11.8) is of the same size (in fact, is the same) as the standard symmetry quotient.

11.6 Experimental Evaluation

In this section we compare the aggregate technique quantitatively with the naive method for verifying bounded parameterized systems, which simply considers all systems individually ("one-by-one"). Experimental results are obtained using BDD-based symbolic model checking. In tables, "N" refers to the parameter bound. We discuss the relationship between the present method and general parameterized model checking approaches in section 11.7.

The one-by-one method and the aggregate technique have the same theoretical power: they can be used to verify arbitrary parameterized systems up to some finite bound. We show experimental results demonstrating the superior efficiency of the aggregate method.

The first example, "McsLock", is the queuing lock algorithm we have seen in previous chapters. This protocol has a global variable that counts processes in the queue (such counters are disallowed by many fully parameterized techniques). It also has a transition that causes several processes to change their local state simultaneously; this transition depends on the number of components in the system. We show in table 11.1 how the aggregate method scales for an increasing number of components. As can be seen, the BDD size for the transition relation R is only slightly bigger than that for R_N. The benefit of the aggregation is to reduce the verification time, which it does by a factor that increases with N.

The second example is a parallel program. Written for a particular cluster of machines, such programs have a natural upper bound on the parameter: the physical number of CPUs in the cluster. Due to the possibility of failures and down-times, such programs are parameterized by the number of available processors. These characteristics make them a suitable application domain for bounded parameterized verification.

We present here a variant of parallel *odd-even sort*, taken from [KGGK94]. This algorithm proceeds in rounds; during even rounds processors compare each even-indexed element they own with the element's right neighbor (which may be owned by the next processor), analogously for odd rounds. The odd-even split ensures mutual exclusion when changing the position of elements. The initial state is unconstrained; the number of elements to be sorted grows with N. The CTL property we verified is of the form $\mathsf{AF}\,\textit{sorted}$.

The Kripke structure derived from this algorithm is asymmetric since the processors have a translational (noncyclic) communication pattern. Because of this irregularity and the liveness-type property, we believe that most existing parameterized techniques are not immediately applicable to automatically verify this algorithm correct for all size instances.

The results in table 11.1 show again clearly the time savings obtained through the aggregate method. In contrast to the McsLock example, the BDD for the aggregate happens to be of a form that allows it to be traversed with fewer live BDD nodes compared with the one-by-one technique. Note that the number of live BDD nodes depends strongly on implementation details in the BDD package. On the other hand, the number of nodes of a particular BDD does not, and indeed the sizes of R_N vs. R are as expected. The differences between R_N and R are bigger than with McsLock since the sorting problem is much less homogeneous—individual transition relations depend a lot on the instance size.

Finally, we present the response of the method to situations in which a property is true for some but not all size instances. The sorting procedure requires comparing each processor's final element to the first of the next processor; the last processor must be treated specially. The parity (even/odd) of the final element owned by each processor alternates if the number of elements per processor is odd. It is easy to get the communication of the boundary cases wrong. Below is the

N	One-by-one method for $n \in [1..N]$			Aggregation method for N		
	BDD Size of R_N	Peak Number of BDD Nodes	Time	BDD Size of R	Peak Number of BDD Nodes	Time
McsLock (N = number of processes):						
5	924	19,165	2.4s	958	19,176	0s
10	2,012	384,449	1:30m	2,057	384,796	53s
15	3,082	1,797,874	39:08m	3,147	1,797,711	15:17m
20	4,173	5,142,717	6:23h	4,346	5,142,890	1:50h
Parallel Sorting (N = number of parallel processors):						
5	962	37,699	3s	2,021	26,106	3s
7	1,614	144,111	52s	3,643	90,249	30s
10	2,881	673,727	21m	6,911	371,529	7m
13	4,450	2,190,163	3:30h	11,129	1,099,196	54m

Table 11.1: Comparison one-by-one and aggregate verification method

output of the method for a version of the algorithm that fails to compare the last two elements of the last processor if the number of processors is odd:

```
Initial states violating "AF sorted" for N=10:
  -   $   $   $   $   $   $   $   $   $
  -   -   -   $   $   $   $   $   $   $
  -   -   -   -   -   $   $   $   $   $
  -   -   -   -   -   -   -   $   $   $
  -   -   -   -   -   -   -   -   -   $
```

Here, '$' represents as before the local state of out-of-bounds processors. The values carried by active processors have been abstracted away and replaced by '-' to more conspicuously expose the delinquent systems: The number of '-' in a global state (i.e. in one row) equals the state's width and thus indicates the parameter size of the system. In our case, these sizes are all odd (1, 3, 5, 7, 9), giving a potentially substantial hint as to where the problem lies.

11.7 Conclusions and Bibliographic Notes

Chapter 11 of this dissertation shows how to collapse a range of instances derived from an arbitrary parameterized system into a single aggregate, which is detailed enough to be able to simulate each instance. Further, initial states of the original systems can be converted appropriately to states of the aggregate, enabling us to verify arbitrary CTL* properties for all instances up to some finite size in one fell swoop. The large time savings obtained in this manner come at little or no additional space cost; the difference is sometimes masked by the fluctuating performance of BDD-based symbolic model checking procedures. As a special case, if the systems are individually symmetric, then so is the aggregate system, which can thus be symmetry-reduced. The aggregate method can be viewed as, instead of symmetry reducing and verifying all systems individually and then combining the result ("does any of them have an error?"), combining the systems first and then applying the reduction and verification once.

We have presented experimental results using a BDD-based implementation of the technique. We believe the method can likewise be used with SAT-based symbolic verification such as Bounded Model Checking (see section 2.3.3, 41); crucial is the ability to operate on sets of states in one step. We remark on the side that despite the common "bounded", the goals of BMC (investigating bounded time lines over a fixed structure) and of the aggregate technique (investigating unbounded time lines over a bounded family of structures) are quite different.

For the presentation of this work, we have made the simplifying assumption that the number of local states, l, is independent of the number of processes, n. This need not be the case. For example, say a particular application requires the processes to form a priority queue, which can be realized by having each process keep a pointer to its successor in the queue. This pointer is part of the local state of the process, such that l roughly equals n. We then need $\log n$ bits to store a local state of a process belonging to M_n. To apply the aggregate method, we use the same technique locally that we used at the system level: we grant every process $\log N$ bits to store its local state, which makes this number independent of n.

Treatment of global variables. Global variables are used for communication and synchronization among processes, and they may appear in atomic propositions of CTL* formulas. Their presence is mostly orthogonal to the present technique. To form the aggregate system M, we distinguish two types of global variables. Those with range independent of the system size n (such as a Boolean semaphore) are introduced into M with the same range. ID-sensitive global variables, i.e. those ranging over process indices and thus with range $[1..n]$ in M_n, are assigned a range of $[1..N]$ in the aggregate structure, equal to their range in structure M_N. An example is the variable *tok* in figure 2.3 earlier. Regarding the definition of *proper*, a variable like *tok* must be restricted to $[1..n]$ in a proper state of width n, despite the variable's range $[1..N]$ in the aggregate. The completion operator leaves the values of all global variables unchanged.

The results of this chapter of the dissertation were first published in [ETW06] and compare with related work as follows. If applicable, successful approaches to parameterized model checking (PMC) (see e.g. [Lub84, GS92, AJ99, APR$^+$01]) have the clear advantage that they show correctness for all sizes. The bounded and unbounded formulations of PMC synergize when unbounded techniques reduce the correctness for infinitely many instances to correctness up to some finite cutoff. This cutoff depends on the communication complexity of the parameterized system and is not guaranteed to be small [EK00, BHV03, CMP04]. The aggregate method can therefore be used as a follow-up to cutoff-based approaches, picking up the task of verifying the remaining finite-size family.

The disadvantage of unbounded methods is that, targeting a generally undecidable problem, a fully automated solution that works for any input system does not exist. Many authors forfeit completeness by imposing restrictions on the input syntax in order to allow an algorithmic solution. In an early work, E. Clarke, O. Grumberg and M. Browne assume the absence of global variables [CGB86], which could be used to distinguish the number of components. The McsLock example discussed above contains such a global counter variable. Counters may also occur in dynamic systems that monitor the number of active components, for instance for performance reasons. Interestingly, consider a dynamic system with an energy-saving mode of operation, which is assumed when the active process count falls below some threshold. If this mode has a bug, the system may be correct for a large number of processes, but not for a small one. E. Clarke, O. Grumberg and M. C. Browne say the following about uniform and nonuniform verification:

> *It is easy to contrive an example in which some pathological behavior only occurs when, say, 100 processes are connected together. ... Nevertheless, one has the feeling that in many cases this kind of intuitive reasoning*[2] *does lead to correct results. The question ... is whether it is possible to provide a solid theoretical basis that will prevent fallacious conclusions in arguments of this type.* [CGB86]

However, ensuring that a particular description language permits uniform verification leads to restrictions in which many systems or properties are inexpressible. Specifically, the property logic used in [CGB86] bans the next-time operator X and arbitrarily nested $\exists$ and $\forall$ quantifiers over process indices, which again—like global variables—can be used to count and thus to "cheat". This makes some natural properties cumbersome to express, such as deadlock reachability [EK02] or even mutual exclusion [CGB86]. In contrast, the method of this chapter—being less ambitious—requires no restrictions on the input syntax, and is valid for full CTL* (and even

[2] i.e. inferring correctness for all from correctness for some—T.W.

the μ-calculus).

Other approaches sacrifice full automation. In [CG87], the notion of a *closure process* is introduced, whose definition depends on the parameterized system at hand to a degree that seems to undermine mechanization. In [KM89], the authors present a fairly broad induction method to reduce a family of systems to a single system, using an *invariant process*, which enforces a partial order among the processes. Finding such an invariant requires help from the designer and can be nontrivial. An advantage of the induction method is that—like the aggregate method, but unlike most other parameterized verification techniques— it can detect cases in which a property is violated for some all instances. The Mur φ tool supports replicated components for fully symmetric systems [ID99]. The tool automatically checks whether the given program allows generalizing the verification result to larger systems. The designer, however, is still left with checking the authenticity of returned error traces. Since the aggregate method is exact, there is no need to solicit human interaction for path-lifting, or other forms of manual assistance.

Some works on parameterized verification make use of the apparent symmetry in systems defined using a single process template. In [EN96] and [EK00], full symmetry of the Kripke structure is exploited by appealing to *state symmetry* [ES96] of the property. In contrast, we show how to take advantage of *internal symmetry* of the property and the Kripke structure through a quotient construction.

Part IV

Conclusions

Chapter 12

Summary of Results

Systems of many concurrent components naively engender intractably large state spaces. They can nevertheless be successfully subject to exhaustive formal verification, provided the components can be classified into a few types.

Most phenotypical of state explosion is that a system of many concurrent components induces a straightforward Kripke model that is orders of magnitude larger than the number of components. This remains true even if there exist only a few component *types*, of which all components are instances. In this case the components of the same type may be identical or very similar. It is then often possible to collapse system states that are identical up to exchanged rôles of components of the same type. This dissertation has contributed reduction and verification techniques for systems of a large number of replicated components:

1. We have demonstrated effective and efficient fundamental solutions for exploiting replication. We were able to not just improve on prior approaches, but in fact render reduction based on replication practicable.

   (a) We delivered the message that compact data structures, such as symbolic notations for Kripke models, are not automatically inflexible and useful only for special purposes. They may support fewer efficient operations than explicit, uncompressed data structures. But for many basic problems in computer science, there are many solutions, and some of them may be designed in a way that makes them compatible with rigid formalisms such as binary decision diagrams. We note, however, that the solution in chapter 5 required rather low-level manipulations, namely at the BDD graph level. The point of that approach is that the symmetry condition cannot be expressed efficiently at the logical level (orbit problem, section 4.3.3).

(b) We showed an instance of the widely accepted observation that judiciously integrating verification techniques can produce results that are much stronger than the techniques can individually. Counter abstraction, a generally elegant and powerful method, is useful for systems where component behavior is distilled into a sequence of local state changes. Although it is in principle possible to convert component behavior given in a high-level program into this form, the conversion itself creates a blowup that can render the subsequent reduction meaningless. We demonstrated how static analysis techniques can provide the glue needed between the two behavior representations. As a significant side effect, the program notation resulting from counter abstraction is amenable to processing with symbolic data structures.

2. We extended the aforementioned fundamental techniques to more practical and general scenarios than the mathematical definition of symmetry is immediately able to address. Unfortunately, we had to deal with a discontinuity here: small deviations from the strict preconditions of the previous techniques cannot be healed by equally small and quick fixes. Instead, the extensions

require some work, not only for the algorithm designer, but also for the model checker that implements the extended algorithms. Arguably, deviating arbitrarily far from the preconditions eventually requires fixes so costly that one can do better without any attempt to exploit regularity of structure.

(a) We introduced a flexible adaptive scheme to exploiting partial symmetry in a system of replicated components some of whose transitions may be restricted by few asymmetric guards. We demonstrated that although the algorithms in chapter 10 are not efficient in every case, they do not have to be. Instead, they should focus on the important frequent cases and may even perform poorly on the others as long as there is a mechanism that can warn the user in those cases.

(b) We extended our algorithms to flexible environments where designs are parameterized by the number of components, which increases reusability. Naturally, this also increases the complexity of the problem, rendering it, in full generality, undecidable. Two classical approaches to dealing with undecidable or high-complexity problems are: (i) to solicit human assistance, which makes it possible to exploit particulars of the system at hand, and (ii) to consider a decidable version of the original problem. The second approach is an instance of the paradigm, "if you cannot solve the problem, change it." In this dissertation we presented a technique that can also be viewed as an instance of this paradigm. We made the problem decidable by restricting the number of components to some finite range. At first this idea seems little exciting, since the problem turns now from an undecidable one into one that is trivially solvable by brute force, by divorcing the family of systems into their individual members. The contribution of chapter 11 is that we can do better than brute force. The method of combining the finite family into a single model is attractive

> since it leverages the power of symbolic data structures: the size of a combined representation can be much smaller than the sum of the sizes of the individual representations. In fact, it can be as small as the size of the single largest one of the objects. In this asymptotic case, the smaller systems are taken care of for free.

All results of this dissertation can be summarized and unified as follows. Systems with replicated components tend to engender large state spaces, which is especially striking since they have a small high-level description. On the other hand, models of such systems often have a very regular—symmetric—structure, which can be exploited to reduce the effort of exhaustive verification. This reduction, however, comes at a cost: we may have to build an abstract system first, or we may have to modify the model checking algorithm to incorporate the reduction steps on-the-fly. These costs have long been considered too high, especially with the compact data structures needed to represent the large systems we deal with today. As a result, the use of symmetry was discouraged in connection with such data representations. This dissertation has presented re-encouraging results that, so it is hoped, help symmetry get back to where it belongs: into the center of attention of the verification community.

Chapter 13

Open Problems and Further Research

In this chapter we discuss unsolved problems that are strongly related to the topics presented so far and could make the solutions more complete. We end by sketching a few endeavors that go beyond this dissertation, but are still part of the grand goal of exploiting replication.

13.1 Open Problems

We have seen, especially in the experimental chapter 9, that dynamic symmetry reduction and counter abstraction are roughly complementary: "specific" approaches, i.e. those that store a system state by listing the local states of all components, can be expected to perform poorly whenever there are a large number of components over a small local state space. This is, in contrast, the ideal scenario for "generic" approaches such as counter abstraction. The rule of thumb "$n \gg l$ vs. $l \gg n$" is a good asymptotic measure to predict the performance of the respective techniques, but insufficient in practice since it hides the impact of constant factors. It would likely be beneficial to have a mechanism that, given a fully symmetric problem, decides statically (and quickly) which of the two approaches can be expected to perform better. Such a mechanism can be used in a tool such as DySyRe to free the user from the unpleasant choice of a particular reduction strategy.[1]

Adaptive symmetry reduction (chapter 10) was presented here as maintaining the set of reachable states. We also remarked that simple next-time properties are not preserved, roughly speaking since the subsumption reduction crudely introduces shortcuts into the reachability graph. The temporal operator "X" is a quantitative one: it specifies what is true after one step. The others are more qualitative in nature, both the temporal ones such as F and G, and the branching ones, A and E. What logic exactly is preserved by subsumption is yet open; we suspect it to be more than reachability. Settling this question is also a matter of changing the algorithm: in its current form, algorithm 8 does not label states with properties found to be true, as an explicit-state CTL model checker would have to. Further, how to implement this algorithm symbolically, using BDDs or SAT, is not immediate since some of the operations used by it may not be efficient with those formalisms. As done with dynamic symmetry reduction, we have to find ways to circumvent such operations or perhaps implement them at a lower level than the logical one, say by BDD graph manipulations.

13.2 Further Research

This dissertation does not deal with interesting individual liveness properties. We have seen the liveness property $\mathsf{AF}\,\mathit{sorted}$ in chapter 11, which states that an array of numbers is eventually sorted, independent of the nondeterministic choices an

[1] The Mur φ model checker also offers several reduction strategies (roughly: unique and multiple representatives) and requires the user to make a choice at the command line.

implementation may make. The property *sorted* is formulated over the whole array of numbers, not over an individual number (which in this example takes the rôle of a component). We have also seen the fully symmetric liveness property $\mathsf{AG}(\exists i T_i \Rightarrow \mathsf{AF}\,\exists j C_j)$. It is also global, since it does not specify progress of an individual process, but only of *some* process, provided *some* process (same or not) is trying. Individual progress of any process is written in the form $\mathsf{AG}\,\forall i : (T_i \Rightarrow \mathsf{AF}\,C_i)$. There are two problems with this formula. The first is that it is not symmetric (see section 4.2). This problem can be fixed as follows:

1. equivalently rewrite the formula as $\forall i : \mathsf{AG}(T_i \Rightarrow \mathsf{AF}\,C_i)$;

2. use symmetry to prove that (1.) holds exactly if $\mathsf{AG}(T_1 \Rightarrow \mathsf{AF}\,C_1)$ (i.e. (1.) instantiated with $i = 1$);

3. use $\mathit{Sym}[2..n]$ as symmetry group to verify $\mathsf{AG}(T_1 \Rightarrow \mathsf{AF}\,C_1)$.

That is, process 1 is no longer considered part of the replication: its variables are treated as global. This slightly diminishes the reduction effect, but turns the formula in (1.) into a symmetric one. The second problem, however, is that the formula in (2.) is false under an asynchronous execution model with a nondeterministic scheduler: given a state satisfying T_1, it is possible to never consider process 1 in the future. This is unrealistic since most schedulers have some fairness conditions built in to them, which prevent such futures.

The temporal logic representations of fairness conditions like $\forall i : \mathsf{GF}\,\mathit{exec}_i$, stating that process i should be infinitely often executed, are themselves asymmetric. Intuitively, the $\forall$ quantifier cannot be pushed directly in front of the exec_i, since this changes the formula. Thus, fairness cannot be dealt with by making the fairness condition part of the formula—this would destroy symmetry. The only way out is to change the verification algorithm and make it select only fair paths. The work by A. Emerson and P. Sistla presents a solution using automata that crawl over an annotated quotient structure and keep track of which process makes a step (this information is normally deliberately blurred in quotient structures) [ES97].

Generally, the combination of symmetry and fairness has not yet been explored to a satisfactory degree. Intuitively, symmetry attempts to anonymize components, while fairness requires us to be particular about them, in order to determine which of them fires how often.

Hardware verification brings with it very specific opportunities and demands that differ too much from the ones assumed in this dissertation for it to be immediately useful in this field. We discuss such opportunities and demands.

Replication exists in many hardware designs, since the number of basic building blocks is limited. For the replication to induce symmetry, the building blocks must be placed in a way that makes them indistinguishable. Fortunately, even this is often the case. For example, *memory arrays* are huge arrangements of memory cells, a relatively simple circuitry representing an addressable location in memory. The surrounding circuit allows the cells to be read and written to, activated and deactivated all in the same way, providing a genuinely and fully symmetric system (see the work by M. Pandey [Pan97]). Nevertheless, we have not found the techniques contributed by this dissertation to be very helpful, for two reasons:

proving symmetry: This cannot be reduced to a simple syntactic check, as it is the case, for example, with protocol modeling languages such as MUR φ. The reason is that hardware descriptions usually have a hierarchical form, rather than one where the symmetry is already factored out for us. For example, a 16-bit memory cell, able to hold 65536 different values, is not described as a flat arrangement of 16 cells. Instead, it is composed of two eight-bit cells, each of which is again composed of two four-bit cells, etc. This notation helps keep the description succinct. Flattening the hierarchy makes the symmetry explicit, but causes an intermediate blowup. A possible solution is to work closely with the hardware description language and extract the symmetry from it.

CTL vs. STE: Dynamic symmetry reduction does not abstract a structure up front, but instead performs an on-demand reduction after every image computation. This is useful for deep explorations of the state space, with respect to some initial state. In hardware verification, on the other hand, we often care about the relationship between the current state and the state of the circuit after one clock cycle. For example, to verify that a memory array architecture works correctly, we want to know whether after one step the output bits contain the data of the cell that is being read, similarly for the write operation. Such properties have the CTL form $\mathsf{AX}\, p$, similar to the next-time expression in *Symbolic Trajectory Evaluation* (STE) [Pan97]. Since only a single transition is considered, a dynamic technique that reduces on the fly is not effective.

Another, in principle more promising, way is to use counter abstraction and indeed build a quotient structure. Since the symmetry is full, and we don't have to deal with intricate phenomena that appear in protocols, such as ID-sensitive global variables, this approach seems ideal. However, the main practical shortcoming of counter abstraction does apply: its sensitivity to the local state space size. Concretely, for memory cells, the number of local states is given by 2^b, where b is the (common) number of bits in each cell. If each cell is to contain just a two-byte integer, we are looking at about 65,000 local states.

Note that the remedies to local state explosion we presented in chapter 7 are unlikely to succeed. The "program" in which the hardware is described provides few control flow mechanisms that restrict the reachability of local states (section 7.2) or the "liveness" of the bits (section 7.3). Every local state, i.e. every cell contents, is naturally reachable, and the bits are always live since the next operation may be a read.

Symmetry is a concept that is exploited, or at least acknowledged, throughout the arts and sciences. Close to home, SAT checkers, constraint-satisfaction solvers and similar tools exploit symmetry by symmetry *breaking* (see for example [DLSM04]). That is, during the search for a satisfying solution, the formula is repeatedly changed by adding blocking clauses that prevent the exploration in subformulas that are symmetric to ones already seen. In contrast, model checkers use symmetry *reduction*: the Kripke structure is never changed; instead, encountered states are reduced to representatives of their equivalent classes. The exact relationship between (a) the notions of symmetry of a propositional formula and of a Kripke structure (see [Rin03]), and (b) symmetry breaking and symmetry reduction is an interesting topic to investigate. In recent years, formal verification has profited immensely from the progress made in building efficient SAT solvers. It is conceivable that we could, again, benefit if the relationship between symmetry reduction and symmetry breaking is tight.

Appendix A

Select Proofs

This chapter contains long and technical proofs that would disrupt the flow of the dissertation too much if included in the main text.

A.1 Proof of Lemma 18

Lemma 18 *Algorithm 3 computes α satisfying equation (5.1).*

Proof: We show termination and partial correctness.

Termination: The argument is essentially the same as for standard Bubble Sort. Every call to $swap(p, p+1, Z_{bad})$ brings the local state of at least one of the components p and $p+1$ closer to its correct position. After at most n^2 swaps, there is no pair $(p, p+1)$ left that violates $p \leq_z p+1$. Thus, Z_{bad} as computed in line 2 (algorithm 3 ($\tau(Z)$)) is empty in every iteration of the **for** loop, Z remains unchanged, and the condition $Z = Z'$ in line 5 ($\alpha(Z)$) is true.

Partial correctness: We use two observations.

(i) When the algorithm terminates, we know that for all values of p, Z_{bad} as computed in line 2 ($\tau(Z)$) is empty. Hence, $Z \subseteq \bigcap_{p<n}\{z : p \leq_z p+1\} = \overline{S}$ (equation (5.3)), so $\alpha(T) \subseteq \overline{S}$ ($\alpha(T)$ is the final value of Z).

(ii) Predicate transformer τ manipulates the set Z by applying transpositions (*swap*) to states in Z. Hence, as an invariant, Z and Z' in algorithm 3 contain the same states up to permutations, and thus so do T and $\alpha(T)$ at the end.[1]

These observations allow us to prove $\alpha(T) = \{\overline{t} \in \overline{S} : \exists t : t \in T : t \equiv \overline{t}\}$ as two inclusions:

$\subseteq$: Consider $\overline{t} \in \alpha(T)$. From (i) we know $\overline{t} \in \overline{S}$. From (ii) we conclude that there exists t in T with $t \equiv \overline{t}$.

$\supset$: Consider $\overline{t} \in \overline{S}$, $t \in T$ such that $t \equiv \overline{t}$. From (ii) we conclude that there exists π such that $\pi(t) \in \alpha(T)$. From (i) we conclude $\pi(t) \in \overline{S}$. Since there is exactly one representative of t in $\overline{S}$, we derive $\pi(t) = \overline{t}$, so $\overline{t} \in \alpha(T)$. □

A.2 Proof of Theorem 28

Theorem 28 *Relation $\approx$ is an equivalence relation on S. Moreover, the quotient structure $\overline{M}$ of M with respect to $\approx$ is bisimilar to M with the canonical bisimulation relation $B := \{(s, [s]) : s \in S\}$ relating a state to its equivalence class under $\approx$.*

Proof: We start by showing that $\sim$ is an equivalence relation on the local state space. Reflexivity and symmetry of $\sim$ follow immediately from properties of equality. For transitivity, $(PC_x, x^1, \ldots, x^m) \sim (PC_y, y^1, \ldots, y^m)$ and $(PC_y, y^1, \ldots, y^m) \sim (PC_z, z^1, \ldots, z^m)$ implies $PC_x = PC_z$. Assume an i such that variable v^i is live at PC_x. From the equivalence of the first two states, we conclude $x^i = y^i$, and from the last two, we conclude $y^i = z^i$, thus $x^i = z^i$. Regarding $\approx$, since both "agreement on all global variables" and $\sim$ are equivalence relations, so is $\approx$.

For the second part, the quotient $\overline{M} = (\overline{S}, \overline{R})$ is defined as $\overline{S} = \{[s] : s \in S\}$ (set of equivalence classes of $\approx$), and $\overline{R} = \{(\overline{s}, \overline{t}) \in \overline{S} \times \overline{S} : \exists s \in \overline{s},\ t \in \overline{t} : (s, t) \in R\}$. Given $s \in S$ and $\overline{s} = [s]$, we have to show two things:

[1] In general, however, α cannot be expressed as a single permutation.

1. Assume t such that $(s, t) \in R$. Then let $\bar{t} = [t]$. t and $\bar{t}$ are related under B. By definition of $\overline{R}$, it follows that $(\bar{s}, \bar{t}) \in \overline{R}$.

2. Assume $\bar{t}$ such that $(\bar{s}, \bar{t}) \in \overline{R}$. Then, by definition of $\overline{R}$, there exist $s' \in \bar{s}$, $t' \in \bar{t}$ such that $(s', t') \in R$. By the semantics of asynchronous execution, this means that s', t' agree on the local states of all processes except one, say p, which possibly changes its local state from $l_p(s')$ to $l_p(t')$. Since $s, s' \in \bar{s}$, they have the same PC value, they agree on all global variables, and further on all local variables of process p (in fact, of all processes) except possibly some dead variables, whose values, by definition, are not used at the current PC. It follows that executing $\mathcal{P}$ from local state s gives the same result t' as executing $\mathcal{P}$ from s', hence $(s, t') \in R$. We can therefore choose $t := t'$ and have $t \in \bar{t}$ and $(s, t) \in R$. □

A.3 Proof of Theorem 32

Theorem 32 *State* $\hat{s} = (s, \mathbb{P})$ *subsumes state* $\hat{t} = (t, \mathbb{Q})$ *exactly if*

1. $i \equiv_{\mathbb{Q}} j \ \Rightarrow\ (i \equiv_{\mathbb{P}} j \vee t_i = t_j)$ *is a tautology, and*

2. $t \in Orbit(\hat{s})$*, i.e. there exists* $\sigma \in \langle \mathbb{P} \rangle$ *such that* $\sigma(s) = t$.

Proof:

$\Rightarrow$: 2. follows from $t \in Orbit(\hat{t}) \subseteq Orbit(\hat{s})$. Regarding 1., consider i, j with $i \equiv_{\mathbb{Q}} j$. If $t_i = t_j$, property 1. is proved. Assume now $t_i \neq t_j$, and let $\pi = (i\ j)$, the transposition of i and j. This permutation is generated by $\mathbb{Q}$, thus $\pi(t) \in Orbit(\hat{t}) \subseteq Orbit(\hat{s})$. Therefore, there exists a permutation $\beta \in \langle \mathbb{P} \rangle$ that satisfies $\beta(s) = \pi(t)$. Combining this with $\sigma(s) = t$ (from 2.), we get $\beta(\sigma^{-}(t)) = \beta(s) = \pi(t)$. Thus, we have found a permutation $\alpha := \beta \circ \sigma^{-}$ that satisfies $\alpha(t) = \pi(t)$ and $\alpha \in \langle \mathbb{P} \rangle$ since $\beta, \sigma \in \langle \mathbb{P} \rangle$. (However, we are not done with the proof since $\alpha(i)$ may still be different from j, so we can not yet conclude i and j are $\mathbb{P}$-equivalent.)

Now consider the sequence i, $\alpha(i)$, $\alpha^2(i)$, $\dots$. Since $\alpha \in \langle\mathbb{P}\rangle$, all these elements belong to the same cell within $\mathbb{P}$. The goal is now to show that j is part of this sequence, which shows that $i \equiv_{\mathbb{P}} j$ and thus completes the proof.

Due to the finite domain, the sequence contains a repetition, and due to α's bijectivity, there is in fact an index $x > 0$ such that $i = \alpha^x(i)$. Now consider the local state sequence

$$t_i \ , \quad t_{\alpha(i)} = t_{\pi(i)} = t_j \ , \quad t_{\alpha^2(i)} \ , \quad \dots \ , \quad t_{\alpha^x(i)} = t_i \, .$$

This sequence begins and ends with the same local state t_i. It also contains an element—t_j— that differs from t_i. Thus, there is an index k such that $t_{\alpha^k(i)} \neq t_i$, and $t_{\alpha^{k+1}(i)} = t_i$. $t_{\alpha^k(i)}$ is the $\alpha^k(i)$'th element of t, and $t_{\alpha^{k+1}(i)}$ is the $\alpha^k(i)$'th element of $\alpha(t)$. Recalling that t and $\alpha(t)$ are identical except for positions i and j, and observing that $\alpha^k(i) \neq i$ since t differs at positions i and $\alpha^k(i)$, it follows that $\alpha^k(i) = j$. Hence, j is $\mathbb{P}$-equivalent to i.

$\Leftarrow$: Given are σ (from 2.) and $u \in \mathit{Orbit}(\hat{t})$, i.e. there exists $\pi \in \langle\mathbb{Q}\rangle$ that satisfies $\pi(t) = u$. We have to find a permutation $\alpha \in \langle\mathbb{P}\rangle$ that satisfies $\alpha(s) = u$. A first choice is $\alpha := \pi \circ \sigma$, since then $\alpha(s) = \pi(\sigma(s)) = \pi(t) = u$. However, $\mathbb{P}$ may not generate π and thus may not generate α. We construct a permutation π' with the following requirements:

(i) $\pi'(t) = \pi(t)$, and

(ii) $\pi' \in \langle\mathbb{P}\rangle$.

Once we have that, we choose $\alpha := \pi' \circ \sigma$. Since $\mathbb{P}$ generates both π' and σ, it is $\alpha \in \langle\mathbb{P}\rangle$, and $\alpha(s) = \pi'(\sigma(s)) = \pi'(t) = \pi(t) = u$.

We construct π' incrementally for each cell $Q \in \mathbb{Q}$. We show that for all elements from Q, the required properties are satisfied. Doing this for all $Q \in \mathbb{Q}$ yields the desired permutation on $[1..n]$. We distinguish two cases:

(a) Q is fully contained in some $P \in \mathbb{P}$. In this case, choose $\pi'(i) = \pi(i)$ for all $i \in Q$. Then, on Q, π' is a permutation and satisfies requirements (i) ($t_{\pi'(i)} = t_{\pi(i)}$) and (ii) (since $Q \subseteq P$).

(b) Q is not fully contained in any $P \in \mathbb{P}$. Then there are two elements $i, j \in Q$ such that $i \not\equiv_{\mathbb{P}} j$. By property 1 in the theorem, we conclude that $t_i = t_j$. Now let k be any element of Q. If $k \equiv_{\mathbb{P}} i$, then $k \not\equiv_{\mathbb{P}} j$, thus $t_k = t_j = t_i$. If $k \not\equiv_{\mathbb{P}} i$, then also $t_k = t_i$. In other words, t_k has the same value for any $k \in Q$. We choose π' to be the identity on Q, which fulfills requirements (i) and (ii). □

Appendix B

The Readers-Writers Protocol in DySyRe

To demonstrate the way programs are written in DySyRe, we give the full DySyRe description of a deliberately simple example, an instance of the Readers-Writers problem, in figure B.1. The readers' indices are $[0..(r{-}1)]$, those of the writers are $[r..(n{-}1)]$. There are three local states: N, T, and C. Transitions from N to T and from C to N are unrestricted. A reader ($i < r$) may transit from T to C if no writer currently resides in C ($\forall j \geq r : s_j \neq C$). A writer ($i \geq r$) may transit from T to C if no process currently resides in C ($\forall j : s_j \neq C$). Intuitively, since readers only read, they may enter their critical section at the same time, as long as the writer is outside its own. (This instance resembles the one derived from the synchronization

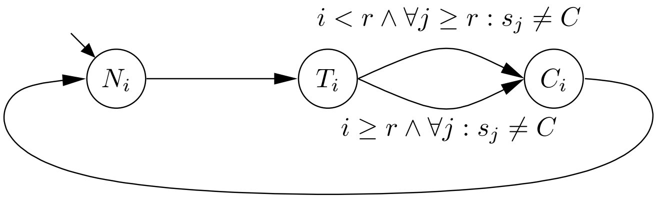

$i < r \wedge \forall j \geq r : s_j \neq C$
N_i
T_i
C_i
$i \geq r \wedge \forall j : s_j \neq C$

Figure B.1: Synchronization skeleton for a version of the Readers-Writers problem

skeleton in figure 10.1 on 141.)

The DySyRe description to realize the Kripke structure derived from this skeleton as a BDD is shown on the next pages; text of the form `// ...` is a comment. To make it more legible, the description is split across different figures. The declaration section of the file is shown in figure B.2. Figure B.3 shows code for auxiliary predicates used in defining the transition relation. The next two figures, B.4 and B.5, show the code that implements the transition relation. Finally, figure B.6 presents the code for the atomic propositions that the user has declared in the beginning of the file and that are thus visible at the property prompt. These are the propositions for the initial states `init` and for the undesirable states `bad`, violating mutual exclusion.

```
// Readers-Writers protocol

Parameter
  r ("number of readers");  // text in parentheses is optional
  w ("number of writers");  // ditto

Const
  READERS = 0;              // process group 0 represents the readers
  WRITERS = 1;              // process group 1 represents the writers

Order
  concat;                   // concatenated variable order

Clique[r]                   // r readers (parameter)
  rstate: enum { N, T, C }; // rstate = local state of reader

Clique[w]                   // w writers (parameter)
  wstate: enum { N, T, C }; // wstate = local state of writer

Proposition
  Init;                     // propositions are only declared here
  Bad;                      // and defined in the Code region below
```

Figure B.2: DySyRe code for Readers-Writers: declarations

Code

```
// An auxiliary predicate: some reader is in local state C.
// Computed as a disjunction over all readers
UBDD exists_reader_C(const StateSpace& S) {
  UBDD result = UBDD::Zero();
  for (ushort p = 0; p < S.r; p++)
    result |= UBDD::equal(S.rstate(p), StateSpace::rstate::C);
  return result; }


// An auxiliary predicate: some writer is in local state C
// Computed as a disjunction over all writers
UBDD exists_writer_C(const StateSpace& S) {
  UBDD result = UBDD::Zero();
  for (ushort p = 0; p < S.w; p++)
    result |= UBDD::equal(S.wstate(p), StateSpace::wstate::C);
  return result; }


// An auxiliary predicate: some process is in local state C
inline UBDD exists_C(const StateSpace& S) {
  return exists_reader_C(S) | exists_writer_C(S); }
```

Figure B.3: DySyRe code for Readers-Writers: auxiliary predicates

```
// Transition relation: disjunction over all transitions
UBDD R(const StateSpace& S) {

  UBDD R = UBDD::Zero();
  UBDD for_p;

  // Readers
  for (ushort p = 0; p < S.r; p++) {
    for_p = UBDD::Zero();

    // N --> T
    for_p |= UBDD::equal(S. rstate(p), StateSpace::rstate::N) &
             UBDD::equal(S._rstate(p), StateSpace::rstate::T);

    // T --> C
    for_p |= UBDD::equal(S. rstate(p), StateSpace::rstate::T) &
             UBDD::equal(S._rstate(p), StateSpace::rstate::C) &
             (! exists_writer_C(S)); // no writer in C

    // C --> N
    for_p |= UBDD::equal(S. rstate(p), StateSpace::rstate::C) &
             UBDD::equal(S._rstate(p), StateSpace::rstate::N);

    // all readers but p and all writers invariant
    for_p &= S.invariant_but(p, S.READERS) &
             S.invariant_sbn(S.WRITERS);

    R |= for_p; }

    // (continued on next page)
```

Figure B.4: DySyRe code for Readers-Writers: Readers' transitions

```
// Writers
for (ushort p = 0; p < S.w; p++) {
  for_p = UBDD::Zero();

  // N --> T
  for_p |= UBDD::equal(S. wstate(p), StateSpace::wstate::N) &
           UBDD::equal(S._wstate(p), StateSpace::wstate::T);

  // T --> C
  for_p |= UBDD::equal(S. wstate(p), StateSpace::wstate::T) &
           UBDD::equal(S._wstate(p), StateSpace::wstate::C) &
           (! exists_C(S)); // no process in C

  // C --> N
  for_p |= UBDD::equal(S. wstate(p), StateSpace::wstate::C) &
           UBDD::equal(S._wstate(p), StateSpace::wstate::N);

  // all writers but p and all readers invariant
  for_p &= S.invariant_but(p, S.WRITERS) &
           S.invariant_sbn(S.READERS);

  R |= for_p; }

return R; }
```

Figure B.5: DySyRe code for Readers-Writers: Writers' transitions

```
// Set of initial states: all processes in N
UBDD Init(const StateSpace& S) {
  UBDD Init = UBDD::One();
  for (ushort p = 0; p < S.r; p++)
    Init &= UBDD::equal(S.rstate(p), StateSpace::rstate::N);
  for (ushort p = 0; p < S.w; p++)
    Init &= UBDD::equal(S.wstate(p), StateSpace::wstate::N);
  return Init; }

// Set of bad states: a writer and some other process in C
UBDD Bad(const StateSpace& S) {
  UBDD Bad = UBDD::Zero();

  // (a) two writers in C:
  for (ushort p = 0; p < S.w; p++)
    for (ushort q = 0; q < S.w; q++)
      if (p != q)
        Bad |= UBDD::equal(S.wstate(p), StateSpace::rstate::C) &
               UBDD::equal(S.wstate(q), StateSpace::rstate::C);

  // (b) a writer and a reader in C:
  for (ushort p = 0; p < S.r; p++)
    for (ushort q = 0; q < S.w; q++)
      Bad |= UBDD::equal(S.rstate(p), StateSpace::rstate::C) &
             UBDD::equal(S.wstate(q), StateSpace::rstate::C);

  return Bad; }

// end of Code region and end of DySyRe input file
```

Figure B.6: DySyRe code for Readers-Writers: atomic propositions

Appendix C

The MCS Queuing Lock Algorithm

In this appendix we list the original code for the list-based queuing lock algorithm from [MS91]. The algorithm achieves mutually exclusive access to a resource using a shared lock, yet the spin instructions that cause a process to wait for access to the resource involve only process-local variables. See [MS91] for a motivation of this feature.

A process is of type `qnode` and has a local variable `locked` and a pointer `next` to the successor in the queue; see figure C.1. The protocol consists of procedures to acquire and release a lock; processes are supposed to call these procedures in alternation (which is not enforced by the protocol). The processes execute based on an asynchronous model of concurrency. Each line in the procedures is considered an atomic transaction.

```
type qnode = record
  next: ^qnode
  locked: bool

type lock = ^qnode

procedure acquire_lock(L: ^lock, I: ^qnode)
  I->next = nil
  predecessor: ^qnode = fetch_and_store(L,I)
  if predecessor != nil            // queue was nonempty
    I->locked := true
    predecessor->next := I
    repeat while I->locked         // spin

procedure release_lock(L: ^lock, I: ^qnode)
  if I->next = nil                 // no known successor
    if compare_and_swap(L, I, nil) // returns true iff it swapped
      return
    repeat while I->next = nil     // spin
  I->next->locked := false
```

Figure C.1: MCS list-based queuing lock [MS91, figure 5]

Bibliography

[AJ99] Parosh Aziz Abdulla and Bengt Jonsson. On the existence of network invariants for verifying parameterized systems. In *Correct System Design (CSD)*, 1999.

[AK86] Krzysztof Apt and Dexter Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters (IPL)*, 1986.

[APR$^+$01] Tamarah Arons, Amir Pnueli, Sitvanit Ruah, Jiazhao Xu, and Lenore Zuck. Parameterized verification with automatically computed inductive assertions. In *Computer-Aided Verification (CAV)*, 2001.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 1999.

[BDH00] Dragan Bosnacki, Dennis Dams, and Leszek Holenderski. Symmetric SPIN. In *Model Checking of Software (SPIN)*, 2000.

[BG02] Sharon Barner and Orna Grumberg. Combining symmetry reduction and under-approximation for symbolic model checking. In *Computer-Aided Verification (CAV)*, 2002.

[BHV03] Ahmed Bouajjani, Peter Habermehl, and Tomás Vojnar. Verification of parametric concurrent systems with prioritized FIFO resource management. In *Concurrency Theory (CONCUR)*, 2003.

[BR00] Thomas Ball and Sriram Rajamani. Bebop: A symbolic model checker for Boolean programs. In *Model Checking of Software (SPIN)*, 2000.

[Bry86] Randy Bryant. Graph-based algorithms for Boolean function manipulation. *Transactions on Computers (TC)*, 1986.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*, 1977.

[CCB$^+$] Roberto Cavada, Alessandro Cimatti, Marco Benedetti, et al. *NuSMV: a New Symbolic Model Checker*. ITC-IRST, Carnegie Mellon University, University of Genova, University of Trento, `http://nusmv.irst.itc.it`.

[CE81] Edmund Clarke and Allen Emerson. The design and synthesis of synchronization skeletons using temporal logic. In *Logic of Programs (LOP)*, 1981.

[CEFJ96] Edmund Clarke, Reinhard Enders, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design (FMSD)*, 1996.

[CEJS98] Edmund Clarke, Allen Emerson, Somesh Jha, and Prasad Sistla. Symmetry reductions in model checking. In *Computer-Aided Verification (CAV)*, 1998.

[CES86] Edmund Clarke, Allen Emerson, and Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *Transactions on Programming Languages and Systems (TOPLAS)*, 1986.

[CG87] Edmund Clarke and Orna Grumberg. Avoiding the state explosion problem in temporal logic model checking. In *Principles of Distributed Computing (PODC)*, 1987.

[CGB86] Edmund Clarke, Orna Grumberg, and Michael Browne. Reasoning about networks with many identical finite-state processes. In *Principles of Distributed Computing (PODC)*, 1986.

[CMP04] Ching-Tsun Chou, Phanindra Mannava, and Seungjoon Park. A simple method for parameterized verification of cache coherence protocols. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2004.

[DDHY92] David Dill, Andreas Drexler, Alan Hu, and Han Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design (ICCD)*, 1992.

[DLSM04] Paul Darga, Mark Liffiton, Karem Sakallah, and Igor Markov. Exploiting structure in symmetry detection for CNF. In *Design Automation Conference (DAC)*, 2004.

[DM05] Alastair Donaldson and Alice Miller. Automatic symmetry detection for model checking using computational group theory. In *Formal Methods (FM)*, 2005.

[DM06] Alastair Donaldson and Alice Miller. Symmetry reduction for probabilistic model checking using generic representatives. In *Automated Technology for Verification and Analysis (ATVA)*, 2006.

[EHT00] Allen Emerson, John Havlicek, and Richard Trefler. Virtual symmetry reduction. In *Logic in Computer Science (LICS)*, 2000.

[EJP97] Allen Emerson, Somesh Jha, and Doron Peled. Combining partial order and symmetry reductions. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 1997.

[EK00] Allen Emerson and Vineet Kahlon. Reducing model checking of the many to the few. In *Computer-Aided Design (CAD)*, 2000.

[EK02] Allen Emerson and Vineet Kahlon. Model checking large-scale and parameterized resource allocation systems. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2002.

[Eme90] Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. MIT Press, 1990.

[EN96] Allen Emerson and Kedar Namjoshi. Automatic verification of parameterized synchronous systems. In *Computer-Aided Verification (CAV)*, 1996.

[ES90] Allen Emerson and Jai Srinivasan. A decidable temporal logic to reason about many processes. In *Principles of Distributed Computing (PODC)*, 1990.

[ES96] Allen Emerson and Prasad Sistla. Symmetry and model checking. *Formal Methods in System Design (FMSD)*, 1996.

[ES97] Allen Emerson and Prasad Sistla. Utilizing symmetry when model-checking under fairness assumptions: An automata-theoretic approach. *Transactions on Programming Languages and Systems (TOPLAS)*, 1997.

[ET99] Allen Emerson and Richard Trefler. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In *Correct Hardware Design and Verification Methods (CHARME)*, 1999.

[ETW06] Allen Emerson, Richard Trefler, and Thomas Wahl. Reducing model checking of the few to the one. In *International Conference on Formal Engineering Methods (ICFEM)*, 2006.

[EW03] Allen Emerson and Thomas Wahl. On combining symmetry reduction and symbolic representation for efficient model checking. In *Correct Hardware Design and Verification Methods (CHARME)*, 2003.

[EW04] Allen Emerson and Thomas Wahl. Efficient reduction techniques for systems with many components. In *Brazilian Symposium on Formal Methods (SBMF)*, 2004.

[EW05] Allen Emerson and Thomas Wahl. Dynamic symmetry reduction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2005.

[FBG03] Jean-Claude Fernandez, Marius Bozga, and Lucian Ghirvu. State space reduction based on live variables analysis. *Science of Computer Programming (SOCP)*, 2003.

[GS92] Steven German and Prasad Sistla. Reasoning about systems with many processes. *Journal of the ACM (JACM)*, 1992.

[HBL+03] Martijn Hendriks, Gerd Behrmann, Kim Guldstrand Larsen, Peter Niebert, and Frits Vaandrager. Adding symmetry reduction to Uppaal. In *Formal Modelling and Analysis of Timed Systems (FORMATS)*, 2003.

[Hol97] Gerard Holzmann. The model checker SPIN. *Transactions of Software Engineering (TOSE)*, 1997.

[HP94] Gerard Holzmann and Doron Peled. An improvement in formal verification. In *Formal Methods for Networked and Distributed Systems (FORTE)*, 1994.

[ID96] Norris Ip and David Dill. Better verification through symmetry. *Formal Methods in System Design (FMSD)*, 1996.

[ID99] Norris Ip and David Dill. Verifying systems with replicated components in Murϕ. *Formal Methods in System Design (FMSD)*, 1999.

[KGGK94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing.* Benjamin/Cummings Publishing, 1994.

[KM89] Robert Kurshan and Kenneth McMillan. A structural induction theorem for processes. In *Principles of Distributed Computing (PODC)*, 1989.

[LS] Shuvendu Lahiri and Sanjit Seshia. *UCLID: A Verification Tool for Infinite-State Systems.* `http://www-2.cs.cmu.edu/~uclid/`.

[Lub84] Boris Lubachevsky. An approach to automating the verification of compact parallel coordination programs. *Acta Informatica*, 1984.

[McM93] Kenneth McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic, 1993.

[McM02] Kenneth McMillan. Applying SAT methods in unbounded symbolic model checking. In *Computer-Aided Verification (CAV)*, 2002.

[MD] Ralph Melton and David Dill. *Murϕ Annotated Reference Manual, rel. 3.1.* `http://verify.stanford.edu/dill/murphi.html`.

[Min01] Shin-ichi Minato. Zero-suppressed BDDs and their applications. *Software Tools for Technology Transfer (STTT)*, 2001.

[MS91] John Mellor-Crummey and Michael Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *Transactions on Computer Systems (TOCS)*, 1991.

[Muc97] Steven Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, 1997.

[Pan97] Manish Pandey. *Formal Verification of Memory Arrays*. PhD thesis, Carnegie Mellon University, ISBN 0-591-64726-5, 1997.

[PD95] Fong Pong and Michel Dubois. A new approach for the verification of cache coherence protocols. *Transactions on Parallel and Distributed Systems (TOPDS)*, 1995.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science (FOCS)*, 1977.

[PXZ02] Amir Pnueli, Jessie Xu, and Leonore Zuck. Liveness with $(0, 1, \infty)$-counter abstraction. In *Computer-Aided Verification (CAV)*, 2002.

[QS82] Jean-Pierre Quielle and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming (ISOP)*, 1982.

[Rin03] Jussi Rintanen. Symmetry reduction for SAT representations of transition systems. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 2003.

[Ros75] Joseph Rosen. *Symmetry Discovered*. Cambridge University Press, 1975.

[Ros95] Joseph Rosen. *Symmetry in Science*. Springer-Verlag, 1995.

[SG04] Prasad Sistla and Patrice Godefroid. Symmetry and reduced symmetry in model checking. *Transactions on Programming Languages and Systems (TOPLAS)*, 2004.

[SGE00] Prasad Sistla, Viktor Gyuris, and Allen Emerson. SMC: a symmetry-based model checker for verification of safety and liveness properties. *Transactions on Software Engineering and Methodology (TOSEM)*, 2000.

[Som] Fabio Somenzi. *The CU Decision Diagram Package, release 2.3.1.* University of Colorado at Boulder,
`http://vlsi.colorado.edu/~fabio/CUDD/`.

[TMGI05] Daijue Tang, Sharad Malik, Aarti Gupta, and Norris Ip. Symmetry reduction in SAT-based model checking. In *Computer-Aided Verification (CAV)*, 2005.

[Wah07] Thomas Wahl. Adaptive symmetry reduction. In *Computer-Aided Verification (CAV)*, 2007.

[WS02] Farn Wang and Karsten Schmidt. Symmetric symbolic safety-analysis of concurrent software with pointer data structures. In *Formal Methods for Networked and Distributed Systems (FORTE)*, 2002.

[Yor00] Karen Yorav. *Exploiting Syntactic Structure for Automatic Verification.* PhD thesis, Technion Israel, 2000,
`http://www.cs.technion.ac.il/users/orna/KarenThesis.ps.gz`.

Vita

(to be added)

Permanent Address: (to be added)

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.