Reducing Model Checking of the Few to the One

E. Allen Emerson[1] Richard J. Trefler[2] Thomas Wahl[1]

[1] Department of Computer Sciences and Computer Engineering Research Center,
The University of Texas, Austin/TX 78712, USA
[2] David R. Cheriton School of Computer Science,
University of Waterloo, Waterloo/Ontario N2L 3G1, Canada

Abstract. Verification of parameterized systems for an arbitrary number of instances is generally undecidable. Existing approaches resort to non-trivial restrictions on the system or lack automation. In practice, applications can often provide a suitable bound on the parameter size. We propose a new technique toward the bounded formulation of parameterized reasoning: how to efficiently verify properties of a family of systems over a *large finite* parameter range. We show how to accomplish this with a single verification run on a model that aggregates the individual instances. Such a run takes significantly less time than if the systems were considered one by one. Our method is applicable to a completely inhomogeneous family of systems, where properties may not even be preserved across instances. In this case the method exposes the parameter values for which the verification fails. If symmetry is present in the systems, it is inherited by the aggregate representation, allowing for verification over a reduced model. Our technique is fully automatic and requires no approximation.

1 Introduction

Model checking is an algorithmic technique for the verification of programs with respect to temporal logic specifications [QS82,CE81]. It is suitable for systems representable by a finite model, which includes many safety-critical applications such as flight-controllers. The method is successfully applied in industry to technical protocols, to computer hardware, and also, more recently, to software.

In practice, many systems are composed of replicated components. Examples include communication and cache coherence protocols, where the components are concurrent processes, and hardware designs, where the components are black-box pieces of logic, such as memory units. To allow for re-usability, descriptions of such systems are usually parameterized by the number of components. The *parameterized verification problem* is to decide whether a given property holds for all (i.e. infinitely many) instances of the size parameter. Due to its broad nature, this problem is generally undecidable [AK86].

[1] Authors supported in part by NSF grants CCR-009-8141 and CCR-020-5483.
[2] Author supported in part by grants from NSERC of Canada and Nortel Networks.
Email addresses: {emerson|wahl}@cs.utexas.edu, trefler@uwaterloo.ca

There are two principle 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 [CGB86,CG87,EK00, see also sections 8 and 9], 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 paper, we propose 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. Of course, 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 non-trivial: 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 in our approach 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 (see section 2.2) 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, our findings 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 suggested method is applicable to an arbitrary, *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 (see comments in [CGB86]; an exception is [KM89]). In contrast, our technique 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 the paper, we consider the special case of *symmetric*, i.e. more homogeneous, families. 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 not enough. None of this is of any concern with our method.

In summary, we view our approach as a supplement to parameterized verification, which is generally intractable. The proposed method 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. We document its efficacy by experimental results in section 8.

2 Background

The following paragraphs contain basic material about symbolic model checking and temporal logics; the reader familiar with these topics is invited to skip ahead to section 3.

2.1 Model Checking and Temporal Logic

Model checking requires that the system under investigation be expressed as a finite-state model, and that the desired properties be written in a temporal logic that is understood by the model checker at hand. Formally, a model M consists, at a minimum, of a finite set of states, S, and a transition relation, R. The set of states is usually obtained as the set of all possible valuations of system variables. R is a relation, in order to allow for non-determinism. Finally, sometimes we also explicitly define a labeling function, L, which provides the "glue" between the model and the properties to be verified: it assigns to each state atomic properties that are true at that state, such as "error state" or "initial state". These *atomic propositions*, forming a set AP, are used as atoms in temporal logic formulas. Summarizing, given a finite set S, we have $R \subseteq S \times S$ and $L\colon S \to 2^{AP}$.

Popular temporal logics used for the specification of program properties are (enhanced versions of) LTL [Pnu77] and CTL [EC82]. Both logics can be thought of as propositional logic extended by operators related to the evolving nature of programs through states. More precisely, LTL features temporal operators such as X, F, and G, which express that their argument is true in the next state, in

some future state, and in all future states, respectively. CTL, on the other hand, has operators characterizing both temporal and branching behavior of programs, such as AX, EF, AG, which express that their argument is true in all successors of the current state, in some future state along some execution path, and in all future states along all execution paths, respectively. For example, the LTL formula GF *executed* states that with respect to the current state, the atomic predicate *executed* is always (G) eventually (F) true, i.e. infinitely often. The CTL formula EF *sorted* states that along some execution path of the program, at some point the predicate *sorted* will be true (we say a *sorted* state is *reachable*). Neither of the two logics subsumes the other; the quite powerful logic CTL* is a superset of LTL ∪ CTL. A formal treatment of these logics is beyond the scope of this paper; plenty of literature is available on these topics [Eme90].

Both the system model and the temporal logic properties are presented to a model checker. Given sufficient resources, the result is either a confirmation of the satisfaction of the property with respect to the model, or a failure, in which case often a counter example can be presented. Assuming the counter example is an undesirable one, it is used in debugging the system.

2.2 Symbolic Model Checking

A phenomenon impacting the usability of model checking in practice is the *state explosion problem*, referring to the fact that the state space of a system is often exponentially larger than its description. One successful approach to combat this problem is *symbolic model checking* [McM93]. The idea is that instead of representing the system model $M = (S, R)$ using sets that enumerate the states and transitions, S and R can also be expressed as boolean formulas. For example, the formula $x = 4 \wedge y = 3$ succinctly represents the set of states where x has the value 4 and y the value 3 (with other variables' values unspecified). A formula over current-state and next-state variables can be used to express the effect of a transition. For instance, $next(x) = x + 1$ represents the assignment $x := x + 1$.

Once states and transitions are modeled as boolean formulas, a data structure is needed to encode these formulas. In its original form [McM93], symbolic model checking was implemented using *binary decision diagrams* (BDDs) [Bry86]; today alternative structures are used as well. BDDs can represent many practically occurring systems succinctly, although for some they are provably unsuitable, for instance those involving non-linear arithmetic. One disadvantage of BDDs is that the degree of conciseness depends on quite a few parameters, many of which can only be determined experimentally. An advantage that makes them blend nicely with model checking is canonicity: for a fixed set of parameters, every propositional formula has a unique BDD representation. This facilitates termination detection in model checking routines.

3 Preliminaries

The parameterized systems we consider consist of replicated components, i.e. collections of processes whose behavior is described by a single program. The program can have shared variables; each process is characterized at any time by its local state. We present such programs using the graph-like notation of *synchronization skeletons* [CE81]. Local states are shown as nodes in the graph, transitions as edges. As an example, consider a token-ring solution to the n-process Mutual Exclusion problem with a shared variable $tok \in [1..n]$, and the skeleton in figure 1.

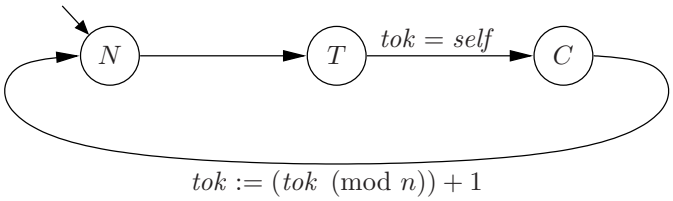

Fig. 1. Synchronization skeleton for a token version of the Mutual Exclusion problem

A skeleton's arcs can be labeled with guards (shown in the figure above the arc) and actions (shown below the arc). Guards are boolean-valued expressions on local states of processes and shared variables. Actions are assignments to shared variables. The actions are executed after the local state change. The skeleton in figure 1 allows a process to enter its critical section C if it currently possesses the token ($tok = self$). Upon leaving C, it passes the token on to the next process.

A synchronization skeleton gives rise to a system of n concurrent processes in the obvious way. To keep the notation simple, we omit shared variables from our state description for now. (Their presence is mostly immaterial to the techniques developed in this paper, as we will discuss in section 9.) A global state s is thus a tuple $(s^1, \ldots, s^n)$ of local states of processes; transitions do not have actions associated with them. Given two states s and t, let the notation $s^i \xrightarrow{g} t^i \in SKEL$ 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. The transition relation R_n of the n-process concurrent system is defined as

$$R_n = \left\{ (s,t) : \exists i : i \leq n : \left(s^i \xrightarrow{g} t^i \in SKEL \ \wedge \ \forall j : j \neq i : s^j = t^j \right) \right\}. \qquad (1)$$

In practice, the behavior of the processes will rarely be given as a synchronization skeleton, but perhaps in a programming language. Deriving a skeleton from a program is fairly straightforward: each valuation of all local process variables defines a local state; local atomic computation of a process (such as assignments to local variables) is abstracted into a single transition.

4 The Aggregate System

The goal of this paper 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. The skeleton gives rise to a family $(M_n)_{n\in\mathbb{N}}$ of Kripke structures with $M_n = (S_n, R_n, L_n)$. With R_n as in (1), we have

$$S_n = [0..(l-1)]^n, \quad R_n \subseteq S_n \times S_n, \quad L_n \colon S_n \to 2^{AP}.$$

Let now N be an integer specifying the maximum number of processes we are interested in, i.e. we consider $n \leq N$. We will 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 1 *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*

$$c(s^1, \ldots, s^n) = (s^1, \ldots, s^n, \underbrace{\$, \ldots, \$}_{N-n}) \in S, \quad c(s_n, t_n) = (c(s_n), c(t_n)) \in R. \quad (2)$$

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 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:

$$R = \bigcup_{n \leq N} c(R_n). \quad (3)$$

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

Corollary 3 *All states along non-empty paths in the aggregate structure M are proper and have the same width.*

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

$$L(s^1,\ldots,s^N) = \begin{cases} L_n(s^1,\ldots,s^n) & \text{if } (s^1,\ldots,s^N) \text{ is proper of some width } n \\ \emptyset & \text{otherwise.} \end{cases} \quad (4)$$

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

5 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 will be that building a BDD for the aggregate R differs only slightly 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 2. 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 2, the number of bits increases by 1 per process, compared with S_N.

Second, how do we implement the transition relation R? Equation (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 4 *Let the family of systems $(S_n, R_n)_{n \le N}$ be given as a synchronization skeleton. Then*

$$\bigcup_{n \le N} c(R_n) = \{(s,t) : s \textit{ is proper of some width } n, \textit{ and } \exists i : i \le n : \left(s^i \xrightarrow{g} t^i \in \mathit{SKEL} \ \wedge \ \forall j : j \ne i : s^j = t^j \right) \} \quad (5)$$

(In the expression $s^i \xrightarrow{g} t^i \in SKEL$, 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 (1), there exists an index i with the property required in (5).

"$\Leftarrow$": Consider (s,t). From (1) and the second line in (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 $(s,t) \in c(R_n)$. □

Discussion. This theorem provides the ingredients for an efficient implementation of R. The left side of equation (5) is identical to the expression defining R in (3). The right side of (5) is almost identical to the right side of (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 source 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 1.

To implement R, we divide the skeleton edges in two classes: those that are independent of the system size n, such as the edge $T \xrightarrow{tok=self} C$ in figure 1, and those that depend on n. 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 figure 2. In the figure, $e(p)$ stands for the propositional formula representing the system 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 8 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.)

1. $R := \emptyset$;
2. **for** $p := 1$ **to** N **do**:
3. **for** every edge e independent of the system size:
4. $R := R \vee e(p)$
5. **for** $n := 1$ **to** N **do**:
6. **for** $p := 1$ **to** n **do**:
7. **for** every edge e dependent on the system size:
8. $R := R \vee (proper(n) \wedge e(p, n))$

Fig. 2. Implementation of the aggregate transition relation R

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 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 $\exists i : h(i) \wedge i \neq \$$.

Finally, consider a system in which no edge depends on the system size. In this case, equation (5) can essentially be replaced by (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 our 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 (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.

6 Verifying the Aggregate System

We are now ready to realize the main goal of this paper: 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 5 *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 (3). (iii) Conversely, consider some $t \in S$ such that $(c(s_n), t) \in R$. By (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

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

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

$$\forall n : n \leq N : M_n, s_n \models f \quad \textit{iff} \quad \forall s : s \in \Sigma : M, s \models f. \tag{6}$$

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

$\Rightarrow$: 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 5.

$\Leftarrow$: 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 5. □

Discussion. Theorem 6 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 (6) is equivalent to $\Sigma \subseteq BDD_f$.

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" (even after completion) 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 (6)) fails. Then there exists a non-empty 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 8 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 $\mathsf{EF}\, bad$ 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.

7 Families of Symmetric Systems

In this section we briefly review *symmetry reduction* in model checking and then demonstrate that the aggregate system inherits contingent symmetry from the individual systems. We conclude by showing how to efficiently exploit the (slightly non-standard) symmetry in the aggregate with literally no change to existing symmetry reduction algorithms.

7.1 Symmetries in Kripke Structures

A Kripke structure $M = (S, R)$ modeling a system of n concurrently executing processes is said to be *(fully) symmetric* if the transition relation R is immune to permutations. More precisely, let Sym_n be the group of permutations on $[1..n]$ and let $\pi \in Sym_n$ act on a state $s \in S$ in the form $\pi(s^1, \ldots, s^n) = (s^{\pi(1)}, \ldots, s^{\pi(n)})$, i.e. by permuting the process indices. Then, M is symmetric if for every $\pi \in Sym_n$ the condition $R = \pi(R)$ holds, i.e. [CEFJ96]

$$(s, t) \in R \quad \text{iff} \quad (\pi(s), \pi(t)) \in R. \tag{7}$$

Intuitively, a system is symmetric if its set of transitions remains invariant when the participating processes are renamed. 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 1 permits only the n rotation permutations and thus inhibits full symmetry. Some conditions can be placed on the skeleton to guarantee that the derived structure is indeed symmetric; see [EW03] for a possible strategy. In this section, we assume such conditions are satisfied.

The *orbit relation* $s \equiv t$ iff $\exists \pi : \pi(s) = t$ is an equivalence relation on the state space; based on it a quotient $\overline{M} = (\overline{S}, \overline{R})$ of M can be constructed in the

usual style of existential abstraction: $\overline{S}$ is a set of unique representatives of the equivalence classes (*orbits*), and

$$\overline{R} = \{(\overline{s}, \overline{t}) : \exists s \equiv \overline{s}, t \equiv \overline{t} : (s,t) \in R\}. \tag{8}$$

Given an appropriate set of atomic propositions that respect the equivalence classes, the quotient turns out to be bisimulation equivalent to the original M. Any CTL* formula over such atomic propositions can thus be verified over the smaller $\overline{M}$ instead of over M. Technical details of symmetry reduction are available in the literature [ES96,CEFJ96].

7.2 Uniformly Symmetric Systems

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 Sym_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 7 *For any $\pi \in Sym_N$ and $s \in S$, define*

$$\pi[s] = \begin{cases} \pi(s) & \text{if } s \text{ is proper of some width } n \\ & \text{and } \forall i : i > n : \pi(i) = i \\ s & \text{otherwise,} \end{cases} \tag{9}$$

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 (9), note that it applies not only to improper states, but also to proper states for which π violates the boundary.

Property 8 *For any $\pi \in Sym_N$ 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 (9), $\pi(i) = i > n$ 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 8 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 9 *The family $(M_n)_{n \leq N}$ of systems is called* uniformly symmetric *if*

$$\forall n : n \leq N : \forall \pi : \pi \in Sym_N : \ \pi[c(R_n)] = c(R_n). \tag{10}$$

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 9 provides a closed formulation of this fact and refers to only a single permutation group, Sym_N. This makes reasoning about uniformly symmetric systems convenient. We point out that in equation (10), permutations $\pi[\cdot]$ act according to equation (9), 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 10 *If $(M_n)_{n \leq N}$ is uniformly symmetric, then M is fully symmetric.*

Proof. Let an arbitrary $\pi \in Sym_N$ be given; we show $\pi[R] = R$:

$$\pi[R] \overset{(3)}{=} \pi \left[\bigcup_{n \leq N} c(R_n) \right] \overset{(*)}{=} \bigcup_{n \leq N} \pi[c(R_n)] \overset{(10)}{=} \bigcup_{n \leq N} c(R_n) \overset{(3)}{=} R,$$

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 (9) 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.

7.3 Symmetry-Reducing the Aggregate System

Looking at the somewhat ungainly equation (9) 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 will show that such 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 $orbit(s)$ with respect to the orbit relation [CEFJ96,ID99]. A common choice for the representative is the orbit's lexicographically least element, $\min_{\text{lex}}(orbit(s))$, given some total order $\leq_L$ on the local states. For example, in a 3-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 (9); instead permutations can be applied in the traditional way, with the same result:

Theorem 11 *Let s be a proper state. Then*

$$\min_{\text{lex}}\{\pi[s] : \pi \in Sym_N\} \;=\; \min_{\text{lex}}\{\pi(s) : \pi \in Sym_N\}. \tag{11}$$

Proof. Let n be the width of s, and let $P_{[s]}$ and $P_{(s)}$ be the two sets in the scope of the $\min_{\text{lex}}$ operator in (11). Then $\min_{\text{lex}} P_{[s]} \geq \min_{\text{lex}} P_{(s)}$ follows from $P_{[s]} \subseteq P_{(s)}$. To see the subset property, consider an element $\pi[s]$. If $\forall i : i > n : \pi(i) = i$, then $\pi[s] = \pi(s) \in P_{(s)}$. If not, then $\pi[s] = s = id(s) \in P_{(s)}$, for the identity permutation $id \in Sym_N$.

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 11 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 (9) is of the same size (in fact, is the same) as the standard symmetry quotient.

8 Applications

In this section we compare our technique with two alternative methods for verifying parameterized systems: the naive method that simply considers all systems individually ("one-by-one"), and general parameterized model checking approaches. Experimental results are obtained using BDD-based symbolic model checking. In tables, "N" refers to the parameter bound. "Peak Number of BDD Nodes" is the maximum number of BDD nodes live at any point during execution and thus is a measure of the memory requirements of the method. Running

times are given in seconds (s), minutes (m), or hours (h), as appropriate. We used the *CUDD* BDD package [Som], with a BDD variable order statically chosen to best-fit each problem. All experiments were performed on a 1.6GHz PC with 512MB of RAM running a variant of the Linux operating system.

8.1 Comparison to the One-by-one Method

The one-by-one method and our 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 superiority of our method in terms of efficiency.

The first example, "McsLock", is a model of a queuing lock algorithm [MS91]. It has a shared 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 1 how our 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 our technique is to reduce the verification time, which it does by more than an order of magnitude for the larger instances, and this factor 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* [KGGK94]. This algorithm proceeds in rounds; during even rounds processors compare each even-indexed element they own with its 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 $\mathsf{AF}\, \mathit{sorted}$.

The results in table 1 show again clearly the time savings obtained through our 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.

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 1. Comparison one-by-one and aggregate verification method

8.2 Comparison to PMC Approaches

If applicable, successful approaches to parameterized model checking (PMC) (see e.g. [Lub84,GS92, many others]) have the clear advantage that they show correctness for all sizes. Interestingly, 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]. Our 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, Clarke, Grumberg and Browne assume the absence of shared variables [CGB86], which could be used to distinguish the number of components. The McsLock example discussed above contains such a shared counter variable. Counters may also occur in dynamic systems that monitor the number of active components, for instance for performance reasons. Interestingly, if an "energy-saving" mode of operation has a bug, the dynamic system may be correct for a large number of processes, but not for a small one.

The logic used in [CGB86] also bans the next-time operator X and arbitrarily nested $\exists$ and $\forall$ quantifiers over processes indices. This makes some natural properties cumbersome to express, such as deadlock reachability [EK02] or even mutual exclusion [CGB86]. In contrast, our method—being less ambitious—requires no restrictions on the input syntax, and is valid for full CTL* (and even 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 non-trivial. The MURφ 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 our method is exact, there is no need to solicit human interaction for path-lifting, or other forms of manual assistance.

Looking back at the parallel sorting example, the Kripke structure derived from this algorithm is asymmetric, since the processors have a translational (non-cyclic) 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.

Finally, we present the response of our 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 output of our 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.

9 Conclusion

In this paper we have shown 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 sometimes being 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. Our 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 our technique. We believe the method can likewise be used with SAT-based symbolic verification such as *Bounded Model Checking* (BMC) [BCCZ99]; crucial is the capability 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 our technique (investigating unbounded time lines over a bounded family of structures) are quite different.

Treatment of shared variables. Shared 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 our techniques. To form the aggregate system M, we distinguish two types of shared 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* shared 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 1 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 shared variables unchanged.

Other related work. In addition to the results on parameterized verification mentioned in section 8, there are some that make use of the apparent symmetry in systems defined using a single process template. Full symmetry of Kripke structures can be exploited using some form of counters [EN96,ID99], or by appealing to *state symmetry* [ES96] of the property [EN96,EK00]. In contrast, we show how to take advantage of *internal symmetry* of the property and the Kripke structure through a quotient construction.

Future Work. A topic for further investigation is which reductions other than symmetry are preserved during the aggregation. This seems promising since the aggregate *faithfully* simulates the individuals. The success will depend on how much existing reduction algorithms have to be adjusted to work on the aggregate, and how much efficiency and compression is lost as a result of such adjustments.

References

[AK86] Krzysztof Apt and Dexter Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters (IPL)*, 1986.

[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.

[BHV03] Ahmed Bouajjani, Peter Habermehl, and Tomás Vojnar. Verification of parametric concurrent systems with prioritized FIFO resource management. In *Concurrency Theory (CONCUR)*, 2003.

[Bry86] Randy Bryant. Graph-based algorithms for Boolean function manipulation. *Transactions on Computers (TC)*, 1986.

[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.

[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.

[EC82] Allen Emerson and Edmund Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming (SOCP)*, 1982.

[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.

[ES96] Allen Emerson and Prasad Sistla. Symmetry and model checking. *Formal Methods in System Design (FMSD)*, 1996.

[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.

[GS92] Steven German and Prasad Sistla. Reasoning about systems with many processes. *Journal of the ACM (JACM)*, 1992.

[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.

[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.

[MS91] John Mellor-Crummey and Michael Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *Transactions on Computer Systems (TOCS)*, 1991.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science (FOCS)*, 1977.

[QS82] Jean-Pierre Quielle and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming (ISOP)*, 1982.

[Som] Fabio Somenzi. *The CU Decision Diagram Package, release 2.3.1.* University of Colorado at Boulder, `http://vlsi.colorado.edu/~fabio/CUDD/`.