Available online at www.sciencedirect.com
SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 130 (2005) 379–399
www.elsevier.com/locate/entcs

Efficient Reduction Techniques for Systems with Many Components

E. Allen Emerson[a,1,2] and Thomas Wahl[a,1,3]

[a] *Department of Computer Sciences and Computer Engineering Research Center The University of Texas, Austin/TX, 78712, USA*

Abstract

We present an improved approach to verifying systems involving many copies of a few kinds of components. Replication of this type occurs frequently in practice and is regarded a major source of state explosion during temporal logic model checking. Our solution makes use of symmetry reduction through counter abstraction. The efficiency of this approach directly depends on the size of the components' local state space, which is exponential in the number of local variables. We show how program analysis can significantly reduce the local state space and can help towards a succinct BDD representation of the system. Our reduction techniques synergistically combine into efficient symbolic verification, as documented by promising experimental results.

Keywords: Model Checking, BDD-based Symbolic Representation, Replication, Counter Abstraction

1 Introduction

We consider systems comprised of many copies of a few basic building blocks. Examples include collections of concurrent processes executing the same program, but also compositions of distinct kinds of homogeneous subsystems, such as the readers-writers protocol. Systems of this type are a primary source of *state explosion* during temporal logic model checking, characterized by the potential to incur a state graph much larger than the description of the system.

[1] This work was supported in part by NSF grants CCR-009-8141 and CCR-020-5483, and SRC contract 2002-TJ-1026.

[2] Email: emerson@cs.utexas.edu

[3] Email: wahl@cs.utexas.edu

doi:10.1016/j.entcs.2005.03.019

In this paper, we show how program analysis techniques can significantly enhance the benefit of *symmetry reduction* to limit this problem. A system is *symmetric* with respect to a group of bijections on the state space if the transition relation R is invariant under each bijection π, i.e. if R equals $\{(\pi(s), \pi(t)) : (s, t) \in R\}$. The bijections amount to permutations of the processes. States that are identical up to such permutations are considered equivalent. The quotient structure that is derived naturally from this equivalence is bisimilar to the original structure. Of particular interest is the case of *full symmetry*. Here, the above condition on R is satisfied for every permutation π. Such symmetries allow up to an exponential reduction in structure size. Also, they occur frequently in practice, especially in connection with systems of replicated homogeneous process components.

Model checking on the quotient structure requires an efficient way of determining whether two states are equivalent. This so-called *orbit problem* turned out to be the bottleneck of symmetry reduction, particularly for symbolic representation using binary decision diagrams. The BDD that contains pairs of symmetry-equivalent states is provably of intractable size and must therefore be avoided [2].

One solution that applies to the case of fully symmetric systems is *counter abstraction* [12,9]. It is based on the observation that two global states, viewed as vectors of local states of processes, are equivalent under arbitrary permutations exactly if for every local state L, the number of occurrences of L is the same in the two states. For example, the three states (A, A, B), (A, B, A) and (B, A, A) are equivalent under full symmetry, since in all of them, two of the processes reside in local state A, one in B. The states can be represented succinctly as the tuple $(2A, 1B)$ of counters. The approach not only avoids the orbit problem, but also reduces a system of size l^n (n processes, each with l possible local states) to one of size roughly n^l (l counters, each with range $[0..n]$). With respect to the number of processes, the system size has been reduced from exponential to polynomial [6,7].

Unfortunately, the *local state explosion* problem, encountered often in practice, can have a negative impact on these benefits. A local state of a process is a valuation of the process' local variables. For example, each of the 2^m assignments of values to m boolean local variables forms a local state. The size of the local state space is thus exponential in the number of local variables. Introducing a counter for every local state becomes infeasible in connection with symbolic data structures like BDDs, which require bits to be reserved a priori for each counter.

One objective of this paper is to limit local state explosion. We show that an analysis of the input program describing the processes' behavior often reveals opportunities to reduce the number of local states that actually must be monitored. Although in principal a local state is defined by the values of all local variables, it is sufficient to restrict attention to those that are *live* at certain points in the program, i.e. whose current value is used along some future path. Experience shows that for many programs, only a fraction of their variables are live at any time during execution. We go on to demonstrate instances of static *local reachability* analysis that can further cut down on the size of the abstract system: The counter of an unreachable local state is invariably 0 and does therefore not have to be introduced into the counter-abstracted model. Many cases of unreachable local states can be detected through an over-approximation of the local state space of each process or by statically analyzing the process' program, before building the global Kripke model.

Live variable analysis is frequently applied in compiler optimization to improve run time performance. A key contribution of this paper is to show a way in which it can be very beneficial for verification as well, namely through a potentially exponential reduction of the size of the counter-abstracted model. In the best case, if no two of m boolean local variables are ever live together, the abstract model needs only 2 counters per program control point, despite 2^m conceivable local states.

The techniques can be applied algorithmically. They are efficient, since they operate on the source code, which is usually small compared to a Kripke model. Finally, the proposed reductions are *exact*, i.e. the reduced system has the same behavior as the original one (they are bisimilar).

Languages intended for modeling asynchronous systems often allow changing the local state of many components in one atomic step. For example, a *reset* operation might cause every component to return to its initial state. In the abstract program, this requires a synchronous update of a large number of counter variables. We show how *serialization* can greatly reduce the complexity of symbolically representing such statements, which seem to occur quite frequently in practice. The serialized execution is efficient and preserves all properties of the system expressible in the temporal logic CTL.

Our framework allows symbolic model checking of arbitrary CTL formulas for systems specified in the high-level and flexible modeling language of the Murφ verifier [14]. We exploit symmetry using counter abstraction whenever possible. (Sub-) Systems with full symmetry, as marked by Murφ's *scalarset* type, are converted into a hybrid representation of counters (replacing the symmetric part) and specific state variables.

We structure this paper as follows. Section 2 introduces the model of computation, and provides some background. Section 3 presents the techniques to reduce the local state space. Section 4 describes the serialization of expensive atomic actions. We conclude with experimental results, comparison to related work and future prospects.

2 Preliminaries

Model of Computation.

We assume a system of concurrent process components with an interleaving model of computation. Replicated processes are instantiations of a program template. Any number of templates is allowed; each gives rise to a fully symmetric subsystem. Each process has its own *local* variables, declared in its template. All variables and statements declared outside any template are referred to as *global*. We thus permit compositions of distinct fully symmetric subsystems, which allows us to model systems like *Readers-Writers* (two symmetric clusters) or microprocessors with separate symmetries in channels, memory addresses, registers, etc.

Symmetry Reduction.

Intuitively, the Kripke model $M = (S, R)$ of a system is symmetric if it is invariant under certain transformations π of its state space S. In our case of *process symmetry*, π takes over the task of permuting the processes. Formally, if l_i denotes the local state of process i, $i \in [1..n]$, π is derived from a permutation on $[1..n]$ and acts on a state s as $\pi(s) = \pi(l_1, \ldots, l_n) = (l_{\pi(1)}, \ldots, l_{\pi(n)})$. Given π, we derive a mapping at the transition relation level by defining $\pi(R) = \{(\pi(s), \pi(t)) : (s, t) \in R\}$. Structure M is said to be *fully symmetric* if $\pi(R) = R$ for all π.

The relation $\theta := \{(s, t) : \exists \pi : \pi(s) = t\}$ on S defines an equivalence between states, known as *orbit relation*; the equivalence classes it entails are called *orbits*. It induces a quotient structure $\bar{M} = (\bar{S}, \bar{R})$, where $\bar{S}$ is a chosen set of representatives of the orbits, and $\bar{R}$ is defined as

$$\bar{R} = \{(\bar{s}, \bar{t}) \in \bar{S} \times \bar{S} : \exists s, t \in S : (s, \bar{s}) \in \theta \ \wedge \ (t, \bar{t}) \in \theta \ \wedge \ (s, t) \in R\}. \quad (1)$$

In case of full symmetry, i.e. given $\pi(R) = R$ for all π, $\bar{M}$ is up to exponentially smaller than M and bisimulation equivalent to M; the bisimulation relation is $\xi = (S \times \bar{S}) \cap \theta$. Relation ξ is actually a function and maps state s to the unique representative $\bar{s}$ of its equivalence class under θ.

Properties of systems of concurrent components are usually expressed using indexed atomic propositions, such as C_i, stating that in the given state process i satisfies some property C. To perform model checking on $\bar{M}$, the (maximal) propositional subformulas of the property in question must be invariant under permutations. A permutation acts upon a propositional formula by permuting the indices of the atomic propositions appearing in it. Invariance then means propositional equivalence. For example, the propositional formula $C_1 \vee \ldots \vee C_n$ is invariant under any permutation action. Summarizing, for two states $(s, \bar{s}) \in \xi$ and any formula f over propositional subformulas p such that $p \equiv \pi(p)$ is a tautology for every $\pi \in G$,

$$M, s \models f \quad \Leftrightarrow \quad \bar{M}, \bar{s} \models f. \tag{2}$$

Counter Abstraction.

For BDD-based symbolic verification, symmetry reduction using the orbit relation is likely to be space-inefficient, as shown in detail by Clarke et. al. [2]. An alternative technique makes use of the following observation in order to represent orbits. Two states, i.e. vectors of local state identifiers, are equivalent under θ (identical up to permutation) exactly if for every local state L, the frequency of occurrence of L is the same in the two states—permutations only change the order of elements, not their values. An orbit can therefore be represented as a vector of counters, one for each local state, that records how many of the processes are in the corresponding local state. For example, in a system with local states N, T and C, the states (N, N, T, C), (N, C, T, N), and (T, N, N, C) are all symmetry-equivalent; their orbit (which contains other states as well) can be represented compactly as $(2N, 1T, 1C)$, or just $(2, 1, 1)$.

In practice, it is often possible to rewrite the program describing a fully symmetric system such that its variables are local state counters in the first place (before building a Kripke structure). This procedure is known as counter abstraction, on which this paper concentrates. The advantage of the counter notation is clear: the symmetry is implicit in the representation; the very act of rewriting the program from the *specific* notation of local state variables into the *generic* [6] notation of local state counters implements symmetry reduction. Subsequently, model checking can be applied to the structure derived from the counter-based program without further considerations of symmetry.

Input Language.

We adopted the input language of the Murφ explicit state verifier, because it is widely known, has been used to verify non-trivial examples, and already has a built-in datatype to mark full symmetry. The complete language specification is available from [14]. In brief, transitions in Murφ are known as rules, which may have a guard that must be satisfied for the rule to be enabled, i.e. able to fire. Firing means executing the rule's body, leading to a new state.

The body consists of assignments and high-level statements like loops, subroutine calls, etc. Our verifier accepts a program in this language and performs *symbolic* model checking with respect to CTL specifications (unlike the Murφ tool, which analyzes the reachable state space for invariant violations using an explicit representation of states).

Symmetry is marked in the Murφ input language using the *scalarset* datatype, a symmetric subrange of the integers. Syntactic restrictions guarantee full symmetry of the resulting state graph.[4] For example, a system of n pairwise interchangeable processes can be declared as an array like this: `var proc: array[scalarset(n)] of basetype`, where `basetype` is some user-defined data type that represents local variables. Our verifier recognizes this type of symmetry and ultimately translates the program into an equivalent one, with the specific processes `proc[0..n-1]` replaced by counters.

3 Local State Space Reduction

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.

The number of local states is an important factor for the efficiency of counter abstraction. With BDD-based symbolic model checking, bits need to be explicitly allocated for every counter, whether it is relevant for program execution or not. It is therefore crucial for the performance of counter abstraction to detect situations in which keeping a counter to monitor a local state is unnecessary. Such a situation might arise because some variable values in a local state are unused in the program and hence do not matter (section 3.1), or because the local state is known to be unreachable (section 3.2).

3.1 Live Variable Analysis

We assume the program executed by the processes is specified as sequential code with individual control points that demarcate atomic actions. In this

[4] Compliance with the restrictions is not entirely verified by the compiler, see [14].

case, the local state space of a process contains a program counter, indicating the statement to be executed next. An analysis of 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 3.1 [e.g. [16]] A variable x is **live at a control point** if there exists a path to a future moment[5] at which the value of x is used, and x is not assigned along the path. Otherwise, x is **dead at the control point**.

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

```
0. $nonempty_i := (q > 0);\ q := q + 1$
1. if $nonempty_i$ then
2.     $locked_i := true$                        lock process $i$
3.     wait until $\neg locked_i$                wait for other process to unlock $i$
 { execute restricted code here }                access to some resource, etc.
4. if $q = 1$ then
5.     $q := 0$; goto 0
6. some $j : PC_j = 3 : locked_j := false$       unlock some proc. $j$ waiting at line 3
7. $q := q - 1$; goto 0
```

Fig. 1. Program text for process i

Variable *nonempty* is live only at program line 1. It is used only there, and it is not live before reaching 1, since it is assigned right before in line 0. The consequence is that we do not have to remember the value of *nonempty* at any program line other than 1. For instance, the two local states $(4, \mathit{false}, \mathit{false})$ and $(4, \mathit{true}, \mathit{false})$, written in the order $(PC, \mathit{nonempty}, \mathit{locked})$, do not have to be distinguished, since they differ only in the value of the dead (in line 4) variable *nonempty*. Notice that both local states are otherwise legitimate and in fact *reachable*. A similar argument holds for variable *locked*. It turns out to be live only at line 3, and thus needs to be remembered only at that point of the program. (The occurrence of *locked* in line 6 does not belong to process i.)

How much does this analysis help counter abstraction? The straightforward approach introduces a separate counter for each conceivable local state, of which there are $|[0..7]| \times |\{\mathit{false}, \mathit{true}\}|^2 = 32$. In contrast, following the above analysis, at all lines except 1 and 3, no local variable other than the PC

[5] "Future" is meant to include the present, i.e. the current program line.

matters. For line 1, we only record the value of *nonempty*, and for line 3, only *locked*. The table below lists the local states that the program needs to monitor, using again the notation $(PC, \mathit{nonempty}, \mathit{locked})$ with '$-$' for irrelevant values:

$$\begin{array}{llll}
(0, \ - \ , \ - \ ) & (2, \ - \ , \ - \ ) & (4, \ - \ , \ - \ ) & (7, \ - \ , \ - \ ) \\
(1, \mathit{false}, \ - \ ) & (3, \ - \ , \mathit{false}) & (5, \ - \ , \ - \ ) & \\
(1, \mathit{true}, \ - \ ) & (3, \ - \ , \mathit{true}) & (6, \ - \ , \ - \ ) &
\end{array}$$

We have thus reduced the number of local states to keep track of from 32 to 10.

The formal justification for not recording 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 $\mathbb{P}$ by the processes in an interleaved 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 3.2 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

(i) $PC_x = PC_y$, and

(ii) $x^i = y^i$ 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 $l_p(s)$ and $l_p(t)$ of p in s and t, respectively, satisfy $l_p(s) \sim l_p(t)$.

Theorem 3.3 *Relation $\approx$ is an equivalence relation on S. Moreover, the quotient structure $\bar{M}$ 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. For the first part, 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 $\bar{M} = (\bar{S}, \bar{R})$ is defined as $\bar{S} = \{[s] : s \in S\}$

(set of equivalence classes of $\approx$), and $\bar{R} = \{(\bar{s}, \bar{t}) \in \bar{S} \times \bar{S} : \exists s \in \bar{s},\ t \in \bar{t} : (s,t) \in R\}$. Given $s \in S$ and $\bar{s} = [s]$, we have to show two things:

(i) Assume t such that $(s,t) \in R$. Then let $\bar{t} = [t]$. t and $\bar{t}$ are related under B. By definition of $\bar{R}$, it follows that $(\bar{s}, \bar{t}) \in \bar{R}$.

(ii) Assume $\bar{t}$ such that $(\bar{s}, \bar{t}) \in \bar{R}$. Then, by definition of $\bar{R}$, there exist $s' \in \bar{s}$, $t' \in \bar{t}$ such that $(s', t') \in R$. By the semantics of interleaved 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 $\mathbb{P}$ from local state s gives the same result t' as executing $\mathbb{P}$ from s', hence $(s, t') \in R$. We can therefore choose $t := t'$ and have $t \in \bar{t}$ and $(s,t) \in R$.

□

Counter abstraction of the reduced structure $\bar{M}$ can be implemented fully automatically, and without first building $\bar{M}$, as follows. Determining live variables is a *data flow analysis* problem. A variety of solutions exist, of a complexity that is in practice usually low-degree polynomial in the size of the input program; see for example [16]. The result is, for each value of the program counter, a list of the variables that are live just before 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.

3.2 Local Reachability Analysis

Suppose L is a local state (i.e. a valuation of local variables) that is not reachable 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, and the translation into counters does not have to consider L.

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. However, in order to perform counter abstraction, we do not need to know the exact set of reachable local states; any over-approximation suffices. In fact, not performing such an analysis at all is tantamount to using *all* (conceivable) local states in creating counters. The better we approximate the precise 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. Guards that express dependencies 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 (they have *positive polarity*), and by *false* otherwise. For example, the guard $\exists i : A_i$ is replaced by $A_1 \vee true$ and hence by *true*, whereas $\forall i : B_i$ is replaced by $B_1 \wedge true$ and hence by B_1. Guards on global variables are likewise replaced by *true* or *false*, depending on their polarity. Assignments to global variables are discarded. Essentially, local information of other processes and global variables are viewed as part of an unpredictable environment. The result is a Kripke structure that over-approximates the behavior of a process. 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. Every local state reachable in the global structure is also reachable in the local structure.

Another technique to approximate reachable local states is borrowed from compilers, which sometimes optimize program behavior by confining the number of values that a local variable can have at some program point. Local states not satisfying these limits are unreachable. Examples for such techniques are constant propagation, constant folding, copy propagation, integer interval arithmetic and perhaps even alias analysis (depending on the expressive power of the programming language).[6] Consider the following contrived program, which prints an input number a in some numerical base and the character with ASCII code a, denoted by $chr(a)$.

```
0. **const** $minprint := 32$                                  least printable ASCII code
1. $base := 16$                                                 choose a numerical base
2. **read** $a$
3. **if** $minprint \leq a < minprint + 96$ **then**
4.     **print** $convert(a, base)$, ": ", $chr(a)$           if $a$ among 96 printable characters
```

Variables *minprint* and *base* degenerate to constants, since there is only one (dynamic) assignment to them. After replacing every occurrence by 32 or 16, resp. (constant propagation), these variables do not participate in the construction of local states. More interestingly, in line 4 we know after constant

[6] See for instance [16] for a taxonomy and precise definitions of these techniques.

folding that a satisfies $32 \leq a < 128$, so local states with $PC = 4$ and $a < 32$ or $a \geq 128$ are unreachable. Assuming $\mathit{base} \in \{2, 8, 10, 16\}$ (binary, octal, etc.) and $a \in [0..255]$, the conceivable state space of the above program with local variables PC, base and a has size $5 \cdot 4 \cdot 256 = 5120$. Using the above observations and the fact that a is live only at lines 3 and 4, we obtain a number of counters that need to be introduced of only $1 + 1 + 1 + 256 + 96 = 355$ (one term for each program line).

More generally, regarding the potential of these observations, note that if we have shown for only a single local variable that it cannot assume a particular value at some program line, the total number of local states is reduced by at least 2^{m-1}, given m local variables. (However, for several variables at the same program line, the respective sets of local states eliminated may not be disjoint.)

Discussion.

Both analyses presented in section 3 are performed efficiently on the source code of the program. As described here, both techniques are *static*, i.e. they do not require (partial) execution of the program and ignore communication between components. Instead, they exploit modularity. This makes them a fast front-end to counter abstraction. Another point to note is that live variable analysis (section 3.1) 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 is non-deterministically chosen. In contrast, local reachability analysis via the local Kripke structure (section 3.2) is fit for any input model.

The overall benefit of counter abstraction is dependent 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 $\mathcal{O}(n^l)$, as compared to $\mathcal{O}(l^n)$ 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. For space-efficiency, an explicit-state model checker may use a compressed notation for all zero-valued counters. Symbolically, *zero-suppressed* BDDs [15] may be applicable, but this technique does not capture the benefits of live variable analysis, where the counters shown to be irrelevant are *non-zero*. Moreover, 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 in this section is that they reduce the number of counters before even building a symbolic model; irrelevant ones are simply not present.

4 Serializing Synchronization Constructs

In modeling languages intended for describing asynchronous systems, the granularity of interleaving is determined by whatever the programmer puts inside an atomic action. Such languages therefore usually support constructs that change the local state of several processes at the same time. Common examples are broadcasts to all processes in a symmetric subsystem instructing them to reset their local state in order to recover from a deadlock, or to invalidate their cache data. We call such statements *synchronization constructs.* We show that the straightforward way to implement them abstractly using counters can lead to complex BDDs, and describe an alternative that allows for a more efficient solution.

4.1 Straightforward Counter Abstraction

In this section, we denote the value of local variable x in local state L by $x(L)$ ("x in L"). Assume, for the purpose of an example, every process has a local boolean variable x, and consider the statement (before counter abstraction)

$$\textbf{for } i : x_i := \textit{false}. \tag{3}$$

It changes the local state of all processes i where currently $x_i = \textit{true}$. The straightforward translation of this simple statement into one based on counters is rather involved. For all local states L with $x(L) = \textit{true}$, counter n_L must be set to zero (no process with $x = \textit{true}$ exists after the execution of (3)). Further, for local states L with $x(L) = \textit{false}$, let $L' := L\big|_{x=\textit{true}}$ denote the local state identical to L except that $x(L') = \textit{true}$. Counter n_L increases by $n_{L'}$, since all processes in L' transit to L. These two steps can be implemented by executing

$$\textbf{for } L : x(L) = \textit{true} : \textbf{do } n_L := 0 \tag{4}$$

$$\textbf{for } L : x(L) = \textit{false} : \textbf{do } n_L := n_L + n_{L'} \text{ with } L' := L\big|_{x=\textit{true}} \tag{5}$$

in parallel, or sequentially in the order (5), (4). Parameter L in statement (5) ranges over almost all possible local states, namely all where x is fixed to be *false*. In the worst case, the number of choices for L is thus exponential in the number of local variables other than x. For all such choices of L, the counter addition operation in (5) must be modeled symbolically. Even after the reductions described in the previous section, there is an evident potential for blow-up in the representation of statement (5) as a BDD. Things get worse if variable x has a range V_x of cardinality greater than 2. The choice of the "source state" L' in (5) (which processes leave in order to enter L) is then not uniquely determined any more. In general, for each L, there are $|V_x| - 1$ possibilities for L'.

The intuition for this complexity is an artifact of counter abstraction. In an assignment like **for** $i : x_i := \mathit{false}$, the current value of x_i is overwritten and therefore normally not of further interest. With counter abstraction, however, we need to know this value since the counter for the future local state increases by the value of the counter for the current local state (L vs. L' in (5)). The solution to avoid the complexity is to disentangle steps (4) and (5) so as to execute the original statement (3) in a serial fashion. The key is that this can be done in a way that preserves all properties of the program.

4.2 Enforcing Serialized Execution

Compound statements of the form **for** $i : \mathbf{stmt}_i$ are often such that the result does not depend on the order in which the individual $\mathbf{stmt}_i$ are executed. This is guaranteed, for example, if i ranges over the process indices of a fully symmetric (sub)system. To implement **for** $i : \mathbf{stmt}_i$ serially, execution switches to a mode of operation in which the only enabled statement is $\mathbf{stmt}_i$; we call it the *serial mode*. It will be turned off again once every process has executed $\mathbf{stmt}_i$—the order is irrelevant. To see how such a mode can be enforced, we first turn our attention to a subclass of statements. In the following, we view a statement as a mapping from states to states in the form **stmt**: $S \to S$.

Definition 4.1 A statement **stmt** is called **idempotent** if $\mathbf{stmt}^2 = \mathbf{stmt}$, i.e. executing it twice has the same effect on any state as executing it once.

Observation 4.2 *Statement* **stmt** *is idempotent exactly if for all states* s, $\mathbf{stmt}(s) \in p$ *for* **stmt***'s* fixpoint predicate $p = \{s \in S : \mathbf{stmt}(s) = s\}$.

Thus, executing an idempotent **stmt** produces a state satisfying p, and every state satisfying p is unchanged by **stmt**. All assignments $x := \mathit{expr}$ such that x does not appear in expr are idempotent; the fixpoint predicate is $x = \mathit{expr}$. For conditional statements **if** cond **then** $x := \mathit{expr}$, the fixpoint predicate is $(\neg \mathit{cond}) \vee (x = \mathit{expr})$.

Consider now a statement **for** $i : \mathbf{stmt}_i$ with idempotent $\mathbf{stmt}_i$, and let p_i be $\mathbf{stmt}_i$'s fixpoint predicate. For an intermediate state s during serial mode, the question whether some process j still needs to execute $\mathbf{stmt}_j$ can be resolved using p_j: the answer is *yes* exactly if p_j is *false* at s. The serial mode must therefore be maintained as long as not all p_i are true. More precisely,

(i) let b be a fresh global boolean variable, initial value *false*

(ii) the statement **for** $i : \mathbf{stmt}_i$ is replaced by $b := \exists i : \neg p_i$

(iii) the guard of every existing statement in the program (including the new one in number ii) is strengthened by the conjunct $\neg b$

(iv) the following guarded statement is added to the program, for any process j:

$$b \wedge \neg p_j \longrightarrow \{\ \mathbf{stmt}_j;\ b := \exists i : \neg p_i\ \}$$

This translation procedure is applied to all **for** statements with an idempotent body. One new bit is introduced for each such **for** statement, resulting in a vector $\boldsymbol{b}$ of new variables. The old and the new program give rise to Kripke structures M and M', respectively.

Property equivalence of M and M'.

Let B denote the disjunction of all bits in $\boldsymbol{b}$; this evaluates to *true* over a global state in M' exactly if M' is currently executing one of M's **for** statements. First we observe that M' is neither an over- nor an under-approximation of M, since some behavior was removed from M, other behavior was added. It turns out, however, that all properties written in the temporal logic CTL[7] are preserved if we *ignore* states of M' in serial mode in the evaluation of CTL formulas:

Theorem 4.3 *Let f be a CTL formula, s a state of M, and let s' be the unique state of M' that satisfies $\boldsymbol{b} = (false, \ldots, false)$ and is identical to s on all other variables ($\boldsymbol{b}$ does not occur in s). Then*

$$M, s \models f \quad \textit{exactly if} \quad M', s' \models f',$$

where f' is recursively defined according to the structure of f:

f *atomic proposition:*	$f' = f$
$f = \neg g :$	$f' = \neg(g')$
$f = g \vee h :$	$f' = g' \vee h'$
$f = \mathsf{AF}\, g :$	$f' = \mathsf{AF}\,(g' \wedge \neg B)$
$f = \mathsf{AG}\, g :$	$f' = \mathsf{AG}\,(g' \vee B)$
$f = \mathsf{A}\,(g\, \mathsf{U}\, h) :$	$f' = \mathsf{A}\,((g' \vee B)\, \mathsf{U}\, (h' \wedge \neg B))$
$f = \mathsf{AX}\, g :$	$f' = \mathsf{AX}\, \mathsf{A}\,(B\, \mathsf{U}\, (g' \wedge \neg B)).$

An analogous result holds for formulas with existential path quantifiers. The proof is accomplished by induction on the structure of f and is omitted

[7] For a full description of this logic, see for example [3].

here. As an example, a safety formula of the form $\mathsf{AG}\, good$, to be evaluated over M, is translated into $\mathsf{AG}\,(good \vee B)$ over M', the liveness property $\mathsf{AG}\,(req \Rightarrow \mathsf{AF}\, ack)$ becomes

$$\mathsf{AG}\,((req \Rightarrow \mathsf{AF}\,(ack \wedge \neg B)) \vee B).$$

One can see that almost all basic modalities are adjusted for verification in M' only by adding a propositional disjunct or conjunct (B). Intuitively, the disjunct B allows invariants to ignore states of M' in serial mode, whereas the conjunct $\neg B$ requires eventualities to in fact become true in states of M' that have a counterpart in M, not in serial mode.

A more complex translation is required for the AX modality (last equation in theorem 4.3). This is no surprise, as the serialization of the **for** statements does not preserve next-time: a single transition is replaced by a path of length $n+1$, for the number n of processes.[8]

Efficiency.

The translated program has no transitions that require all processes to execute an idempotent statement simultaneously. Transitions executed by one process at a time require simple counter increments and decrements by 1, with very efficient BDD implementations. The state space of the new program has as many additional bits as there are idempotent **for** statements in the original program. The new statement $b := \exists i : \neg p_i$ is translated into the abstract statement $b := \exists c : c > 0$, where c ranges over counters for local states satisfying $\neg p$. This statement can be expressed very efficiently with a BDD of size logarithmic in the number of counter variables. The additional guard $\neg b$ increases the BDD size of an individual transition by no more than $\mathcal{O}(1)$ node. (The bit representing b should precede the bits for the processes' local states in the BDD variable ordering.)

Since some transitions in M are replaced by paths of length $n+1$ in M', a fixpoint computation in M' may require about n additional iterations. Our experiments show that this overhead is more than compensated by the gain in efficiency due to the reduced BDD complexity of the transition relation.

Generalization.

The above translation relies on the idempotency condition. A general, and less memory-efficient, solution that handles arbitrary **for** loops over all processes can be obtained by introducing a fresh bit b *local to every process.*

[8] This path is unique up to reordering of execution by the processes. Thus, "$\mathsf{AX}\,\mathsf{A}$" in the last equation of theorem 4.3 can be equivalently replaced by "$\mathsf{AX}\,\mathsf{E}$".

These bits are initially *false*; when the **for** statement needs to be executed, they are set to *true*. Interestingly, setting all bits to *true* again requires a **for** loop over all processes. However, this loop has an idempotent body (b_i := *true*), so the technique presented above can be applied. Then, an arbitrary process is selected whose bit b_i is *true* (= "waiting for execution"), its individual statement is executed, and b_i is set to *false*. An equivalence result similar to theorem 4.3 can be formulated.

5 Experimental Results

In this section we show quantitative results of applying our techniques. Symbolic computations are done with our own model checker *UTOOL*, which uses the CUDD BDD package [19]. It takes a program in the input language of the Murφ explicit-state model checker and performs symbolic verification on it, exploiting symmetry using counter abstraction whenever possible. Our tool is flexible in that it has support for less-than-full symmetries as well.

In tables, "Number of BDD nodes" refers to the peak number of BDD nodes allocated at any time during execution. It represents the memory bottleneck of the verification run. In columns titled "Time", the symbols "s", "m" and "h" stand for seconds, minutes, and hours, respectively. For comparative explicit-state model checking, we used the Murφ verifier as available on the Internet (see [14]). All experiments were run on an i686/1400 Mhz PC with 256MB main memory.

The purpose of the first, introductory, example, is to demonstrate the potential of counter abstraction compared to other reduction methods that utilize symmetry (even without applying our program analysis techniques). We consider the classical scenario of r readers and w writers that compete for access to some data. The problem consists of two fully symmetric subsystems, but the global system is asymmetric (due to the writers' priority). The first algorithm shown in table 1, "Multiple Representatives", refers to applying symmetry reduction to the Kripke structure derived from the original program, without counter abstraction. This technique allows symmetry equivalence classes (orbits) to be represented by several states. The advantage to an approach using unique orbit representatives is that a state of the orbit can be mapped to that representative for which this mapping is most efficient; see [2] for details.[9] The next double column lists the results of applying the Murφ verifier to the counter-abstracted program, i.e. non-symbolically. The third algorithm combines counter abstraction and symbolic representation.

[9] The naive approach using the orbit relation mentioned in section 2 is not exhibited here since it fails already for very small problem instances.

Choice of r, w		Symbolic Multiple Represent.		Explicit-State Counter Abstr.		Symbolic Counter Abstr.	
r	w	BDD nodes	Time	Rules fired	Time	BDD nodes	Time
8	8	19,853	1.4s	11,835	0.6s	1,082	0.0s
10	10	41,333	5.8s	25,784	0.6s	1,082	0.0s
16	16	223,770	108.0s	140,471	0.7s	1,379	0.1s
30	30	2,494,219	1:29m	1,482,854	2.2s	1,379	0.1s
100	100	—	—	159,625,349	162.4s	1,973	0.2s
1000	1000	—	—	—	—	2,864	1.5s

Table 1
Results for the Readers-Writers problem

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 does not make use 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 techniques as those described in this paper.

The second example demonstrates the local state space reduction based on live variable analysis from section 3. In many synchronization algorithms, processes denied access to some restricted resource wait ("spin"), constantly polling some global variable. According to [13], this can cause performance bottlenecks, and is partially avoidable. We investigated one of the algorithms proposed in [13] (the list-based spin-lock without atomic compare-and-swap operation), verifying that no two processes can acquire the resource at the same time, that there is no deadlock in the system, and that processes will eventually gain access to the resource, once requested. The program executed by the processes is similar to that in figure 1 in section 3.1.

The table teaches an important 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.

	Symbolic Counter Abstraction …			
	without Reductions		with Local State Reduction	
Processes	BDD nodes	Time	BDD nodes	Time
5	16,583	1.4s	3,022	0.1s
10	71,224	28.8s	9,366	0.6s
15	156,421	110.4s	17,070	1.5s
30	785,411	25:53m	68,332	19.4s
50	2,643,540	3:34h	207,370	145.9s
70	5,586,017	12:16h	454,360	10:18m

Table 2
Results for the MCS Spin Lock algorithm

The final example illustrates the effect of serializing complex synchronous instructions when working with BDDs. A communication bridge transports data between two ports, performing some operations on them in the middle [18]. Processes read the data from the output port. When the output port is full, and no process is ready to consume the data, the system is in a deadlock-like situation. It recovers from it by instructing all processes to interrupt and be ready to unburden the output port. The message is broadcast to all processes, rather than just sent to one, since the output port would likely be full again quickly if only one data item was read from it.

We verified that no data is overwritten at the ports, and that the ports are cleared within a fixed number of steps. The latter property is important for performance guarantees when the bridge is part of a system that is embedded in other devices. The subsystem formed by the processes is, for the purpose of verifying these properties, fully symmetric. Table 3 shows the results for the straightforward counter abstraction of the processes ("simultaneous broadcast"), and for counter abstraction with serialized execution of the broadcast statement.

	Symbolic Counter Abstraction …					
	with simultaneous broadcast			with serialized broadcast		
Processes	BDD size trans. rel.	Total BDD nodes	Time	BDD size trans. rel.	Total BDD nodes	Time
4	10,914	49,576	2.2s	4,177	21,699	2.1s
8	15,850	188,690	10.4s	5,190	66,864	7.7s
16	20,786	830,610	127.4s	6,203	209,157	55.3s
32	25,722	2,683,800	26:43m	7,216	586,938	8:30m
64	30,658	7,310,979	3:36h	8,229	1,955,509	1:12h

Table 3
Results for Communication Bridge example

The figures in the table first tell us that the size of the transition relation was reduced by a factor that grows with the size of the problem. This translates into a reduction of the total number of BDD nodes, to the extent of about the same factor. It is also worth noting that the time savings achieved are slightly less than the space savings. One possible reason was hinted at earlier: The serialization requires more iterations during fixpoint computations to converge.

6 Conclusion

We have shown how to integrate program analysis techniques into symmetry reduction based on counter abstraction for exact and efficient model checking, using symbolic representation with BDDs. Our method is most powerful when the given system contains (multiple kinds of) identical components.

Our approach appears to be unique in its improvement of counter abstraction using program analysis. Recently, [8], [20] and [21] suggest static analysis techniques to optimize BDDs in a concrete (rather than counter abstracted) scenario. The potential savings come from choosing dummy values for dead variables, or from non-deterministic 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 [1], 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 our work 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 [1]) allows us to incorporate live variable information into the abstract state representation, by creating local state counters judiciously.

An interesting goal for the future is to apply our methods to a conservative abstraction of an infinite-state system, using *truncated counters*. An approach to doing this with Murφ was presented in [11]. It is orthogonal to the techniques presented here, since truncation reduces the range of counters, but not their number.

Another further direction is to investigate to what extent counters can be used to reduce systems with less than full symmetry, such as rotational symmetry found in the Dining Philosophers example. Our tool currently falls back on other symmetry reduction techniques in such cases. Applying counter abstraction only to fully symmetric systems is still a huge win, since (1) full symmetry is the most frequent symmetry type, (2) it offers greater potential for savings than other symmetries, and (3) other symmetries suffer to a lesser extent from the orbit relation complexity.

Acknowledgments.

We would like to thank the reviewers for their helpful comments regarding the readability of this paper.

References

[1] Ball, T. and S. K. Rajamani, *Bebop: A symbolic model checker for boolean programs*, SPIN Workshop on Model Checking of Software (2000).

[2] Clarke, E. M., R. Enders, T. Filkorn and S. Jha, *Exploiting symmetry in temporal logic model checking*, Formal Methods in System Design (FMSD) (1996).

[3] Emerson, E. A. and E. M. Clarke, *Using branching time temporal logic to synthesize synchronization skeletons*, Science of Computer Programming (SOCP) (1982).

[4] Emerson, E. A. and A. P. Sistla, *Symmetry and model checking*, Formal Methods in System Design (FMSD) (1996).

[5] Emerson, E. A. and J. Srinivasan, *A decidable temporal logic to reason about many processes*, Principles of Distributed Computing (PODC) (1990).

[6] Emerson, E. A. and R. J. Trefler, *From asymmetry to full symmetry: New techniques for symmetry reduction in model checking*, Conference on Correct Hardware Design and Verification Methods (CHARME) (1999).

[7] Emerson, E. A. and T. Wahl, *On combining symmetry reduction and symbolic representation for efficient model checking*, Conference on Correct Hardware Design and Verification Methods (CHARME) (2003).

[8] Fernandez, J., M. Bozga and L. Ghirvu, *State space reduction based on live variables analysis*, Science of Computer Programming (SOCP) (2003).

[9] German, S. M. and A. P. Sistla, *Reasoning about systems with many processes*, Journal of the ACM (1992).

[10] Ip, C. N. and D. L. Dill, *Better verification through symmetry*, Formal Methods in System Design (FMSD) (1996).

[11] Ip, C. N. and D. L. Dill, *Verifying systems with replicated components in Murφ*, Formal Methods in System Design (FMSD) (1999).

[12] Lubachevsky, B. D., *An approach to automating the verification of compact parallel coordination programs*, Acta Informatica (1984).

[13] Mellor-Crummey, J. M. and M. L. Scott, *Algorithms for scalable synchronization on shared-memory multiprocessors*, ACM Transactions on Computer Systems (TOCS) (1991).

[14] Melton, R. and D. L. Dill, "Murφ Annotated Reference Manual, rel. 3.1", http://verify.stanford.edu/dill/murphi.html (1996).

[15] Minato, S., *Zero-suppressed bdds and their applications*, Software Tools for Technology Transfer (STTT) (2001).

[16] Muchnick, S. S., “Advanced Compiler Design & Implementation”, Morgan Kaufmann Publishers, 1997.

[17] Pong, F. and M. Dubois, *A new approach for the verification of cache coherence protocols*, IEEE Transactions on Parallel and Distributed Systems (TOPDS) (1995).

[18] Simon, D. E., “An Embedded Software Primer”, Addison-Wesley, 1999.

[19] Somenzi, F., “The CU Decision Diagram Package, release 2.3.1”, `http://vlsi.colorado.edu/~fabio/CUDD` (2001).

[20] Wang, F. and K. Schmidt, *Symmetric symbolic safety-analysis of concurrent software with pointer data structures*, Formal Techniques for Networked and Distributed Systems (FORTE) (2002).

[21] Yorav, K., “Exploiting Syntactic Structure for Automatic Verification”, Ph.D. thesis, Technion Israel, `http://www.cs.technion.ac.il/users/orna/KarenThesis.ps.gz` (2000).