Static Analysis for Reductions that preserve Temporal Logic

Karen Yorav (Laster) and Orna Grumberg

Computer Science Department
The Technion
Haifa 32000, Israel
e-mail: {laster,orna}@cs.technion.ac.il
phone: 972-4-829327    fax: 972-4-8221128

Abstract. In this paper we present a method that uses static analysis of programs to create reduced models of the programs. Our algorithm examines the control-flow graph of a program (the syntax) and creates a significantly smaller transition system than would have been created otherwise. The smaller transition system is equivalent to the original transition system of the program with respect to CTL*-X specifications.

Our algorithm is based on syntactic manipulation of expressions. This enables us to handle programs with variables over finite as well as infinite domains. Our method can easily be combined with either explicit state or symbolic methods.

An important advantage of using static analysis is that in order to implement our reduction changes are made only to the compiler (which is relatively simple) and there is no need to change the verification tool or the verification algorithm. This enables integration with existing tools at a very low cost.

We used the Murphi verifier to test the amount of reduction achieved by our method. We let Murphi perform a DFS search and compared the sizes of the original and reduced transition systems. The results show that the reduced system is significantly smaller than the original.
1 Introduction

Though there are many different approaches to automatic verification of programs, they are all limited by the space which is available on a given machine. Even small programs may have significantly large models, so that verifying programs which implement solutions to realistic problems is difficult. The situation in which small programs correspond to exponentially large models is known as the state-explosion problem, and this paper presents another approach to handling it.

A program can be described in two levels, a syntactic and a semantic level. On the syntactic level a program is described by a control-flow graph which is a data-structure that represents its syntactic structure. In this graph each atomic statement in the program is represented by a node, and the edges represent the possible ways in which control can proceed after executing a statement. For the task of verification, we require a semantic model which represents the set of possible computations of the program. The semantic representation of a program is given by a transition system, which consists of a set of states (combinations of values for the program variables) and a set of transitions (a description of how the program can move between states). The size of a control-flow graph is proportional to the number of statements in the program. The size of the transition system is much larger since it is also proportional to the number of variables and the size of their domains. It is the size of the transition system which creates the difficulty in automatic verification.

In this paper we present a method that uses static analysis of programs to create reduced models of the programs. Our algorithm examines the control-flow graph of a program (the syntax) and creates a significantly smaller transition system then would have been created otherwise. The smaller transition system is equivalent to the original transition system of the program, i.e. a given specification is true in the original transition system if it is true in the reduced one. The specifications we consider are formulas of the logic CTL*-X which is similar to CTL*, but without the next-state operator. The reduced transition system that we create is stuttering equivalent to the original transition system of the program, and therefore the truth of CTL*-X formulas is preserved. Our algorithm is based on syntactic manipulation of expressions, and therefore we allow variables with both finite and infinite domains.

Our method automatically creates a reduced transition system directly out of the syntax of the program (the control-flow graph), instead of working on the transition system. We analyze the control-flow graph and identify paths in it on which a process performs a series of consecutive operations that do not influence the specification. A single transition is created, representing the computation along this path (instead of a series of transitions). The transition system created in this way will have less states and less transitions, but will be equivalent to the original transition system w.r.t the given specification. Since we create the reduced transition system directly out of the control-flow graph, and never build the full transition system, we are able to verify systems that would otherwise be too big to handle.

The advantage of our approach is even more significant when the system is composed of several processes. In such a case, each process is reduced separately and only then they are composed to one transition system. This solution thus serves to reduce the exponential blow-up that occurs when taking the cross product of the transition systems of the individual processes.

An important advantage of using static analysis is that in order to implement our reduction, changes are made only to the compiler (which is relatively simple) and there is no need to change the verification tool or the verification algorithm. This enables integration with existing tools at a very low cost. It also means that the overhead of using our reduction is during the (very short) compilation stage and not in the verification processes. Our method can be used for verification using an explicit representation of the transition system as well as for verification using a BDD [4] representation. In either case, the verification algorithm itself is not changed, it just receives a smaller model.

Our reduction is closely related to partial order reductions [?, ?, 22]. Partial order reductions are methods of reducing the state-space traversed by a state-exploration verification algorithm. These methods are based on the observation that sometimes the specification is not sensitive to the different interleavings of computation sequences belonging to processes running in parallel. A partial order
reduction method restricts the search performed on a transition system to a sub-space such that some possibilities for interleaving between transitions are not considered. Our reduction also has the effect of excluding some of the possible interlavings because when we take a series of operations a process \( p \) does and meld them into a single transition we exclude all possible interlavings in which other processes perform operations in the middle. However, we have also taken several steps and changed them into a single step, which regular partial order reductions do not do.

A similar type of reduction to ours was introduced by Miller and Katz in [19]. Their approach was to eliminate invisible states from the transition system, where invisible states are states for which all the entering transitions cannot influence the specification. Their method constructs the projected visible state space relative to a specification through a DFS traversal that eliminates invisible states. The construction of the visible state space requires a linear traversal of a state graph that is somewhat reduced from the original state graph, but is still larger than the transition system produced. The difference from our approach is that we produce the reduced transition system from the syntactic model of the program (the control-flow graph) and not from the transition system representing it. The syntactic model is significantly smaller since it expands only the program counter and not the program variables, which are the source for the enormous size of the semantic model.

In [13, 15] static analyses are used for partial order reductions. An analysis of the statements in the program ([13]) or of the control-flow graphs of the processes [15] is used to determine the transitions to be traversed (ample sets) and therefore to create a reduced model on which a full state-space exploration is performed. These methods have the advantages of static analysis that were mentioned before. The difference between our reduction and theirs is that our reduction does not only inhibit certain interlavings but also “folds” computation segments into a single step. Another important distinction is that our reduction works for the full CTL*-X specification language whereas the reductions presented in [13, 15] are used only for the subset LTL-X.

Our work is also related to works like [1, 5, 8, 14] that use abstract interpretation techniques [7] to obtain reduced models that preserve subsets of the logic CTL*. Their reductions, however, are not fully automatic. The user must define the abstract domain. Furthermore, these works provide weak preservation while our provides strong preservation.

We used Murphi [21] to test the amount of reduction achieved by our method. Murphi is a tool that performs a DFS or BFS traversal of the reachable state space of a program. Murphi programs are a collection of guarded commands, where each command has an enabling condition and a code to be executed when the command is chosen. We chose an example program and translated it into Murphi. We then constructed Murphi descriptions of the reduced systems created by our method using two different specifications. We used Murphi’s DFS search to compare the sizes of the original and reduced transition systems and the time it takes to traverse them. The results show that the reduced system is significantly smaller than the original. In some cases the reduced reachable state space is about 8% of the original one.

The paper is organized as follows. Section 2 gives some preliminary definitions which are needed later on. Section 3 presents a reduction for sequential programs and Section 4 extends the reduction for parallel programs. Section 5 explains how our reduction can easily be integrated with several different verification techniques. Finally, Section 6 gives the results of using our reduction on an example, and Section 7 presents some directions for future research.

2 Preliminaries

2.1 Non-deterministic While Programs

We start by defining a language for sequential programs. It is designed to be simple, and yet include all the essential programming constructs so that it has similar power as “real” programming languages.

\[
P \rightarrow \quad \text{skip} \\
\quad \text{let } x := e \text{ in } p
\]
where $x$ is a program variable, $expr_i$ are expressions over program variables, and $B$ is a boolean condition. The statement $x := \{expr_1, \ldots, expr_n\}$ is a non-deterministic assignment, after which $x$ will contain the value of one of the expressions $expr_1, \ldots, expr_n$. This construct is added to allow the simulation of an input command. Each statement may have a label, identifying the program counter location associated with that statement.

We do not require that the variables be of a finite type, and therefore we allow types such as integers, arrays, and queues.

### 2.2 Control-Flow Graphs

**Definition 2.1:** Given a (sequential) program $P$, its control-flow graph is a graph $CF_P = \langle N, E \rangle$ where $N$ is the set of nodes and $E$ is the set of edges. Each node is associated with a program counter location and the edges represent the possible transitions between locations.

- A node representing a simple assignment or a “skip” statement is labeled with that statement and has a single out-going edge pointing to the node of the next statement to be executed.
- A node representing a non-deterministic assignment $x := \{expr_1, \ldots, expr_k\}$ is labeled with that statement and has $n$ outgoing edges labeled with the expressions $expr_1$ through $expr_k$, all pointing to the node of the next statement to be executed.
- A node of an “if” statement is labeled with the boolean condition of the statement and has two out-going edges labeled “true” and “false”, pointing to the “then” and “else” statements respectively.
- A node of a “while” command is labeled with the boolean condition of the statement and has two out-going edges labeled “true” and “false”, pointing to the body and the next statement after the while respectively.

Figure 1 gives an example of a program and its control-flow graph.

The above definition differs slightly from the regular definition of control-flow graphs in the definition of the edges leaving a non-deterministic assignment node. Usually there is a single outgoing edge, similar to the case of regular assignments, whereas in our definition there are several edges - one for each expression possibly assigned.

We use *transition systems* to define the semantics of programs. Let $AP$ be a predefined set of atomic propositions, which are boolean expressions over the program variables.

**Definition 2.2:** A transition system is a tuple $\langle S, S_0, Tr, L \rangle$, where

- $S$ is a finite set of states,
- $S_0 \subseteq S$ is a set of initial states,
- $Tr$ is a finite set of transitions such that for each $t \in Tr, t \subseteq S \times S$, and
- $L : S \rightarrow 2^{AP}$ is a labeling function which associates each state with the set of atomic propositions true in that state.

A transition $t$ is said to be enabled in a state $s$ if there exists a state $s'$ such that $(s, s') \in t$. It is sometimes convenient to represent transitions by first order formulas. Let $\mathfrak{x} = (x_1, \ldots, x_n)$ be the vector of program variables, and let $pc$ be the variable for the program counter. We create another set of variables $\mathfrak{y} = (x'_1, \ldots, x'_n)$ and $pc'$ to represent the next state. A transition $t$ will be represented by a formula $t(pc, \mathfrak{x}, pc', \mathfrak{y})$ such that every assignment that satisfies it represents a step $((pc, \mathfrak{x}), (pc', \mathfrak{y})) \in t$. A run of a transition system is an infinite sequence $s_1, s_2, \ldots$ such that $s_1 \in S_0$ and for every $i \geq 1$ there is a transition $t$ such that $(s_i, s_{i+1}) \in t$. 
Every control flow graph \( CF_P = (N, E) \) induces a transition system \( ts(CF_P) = (N \times \Sigma, I, Tr, L) \) where \( \Sigma \) is the set of all assignments to the program variables (without the program counter). The initial states set \( I = \{ n_0 \} \times \Sigma \) where \( n_0 \) is the node at the root of the control-flow graph (the program location at the beginning of the program). Every edge \( (n, n') \in CF_P \) is translated into a transition \( t \subseteq S \times S \) in \( Tr \) reflecting the semantics of the command labeling \( n \). Let \( l \) be the label (program counter location) of \( n \), and \( \ell' \) the label of \( n' \). Then:

- If \( n \) is an assignment \( x := \epsilon \) then \( t \triangleq (pc = l) \land (pc' = \ell') \land (x' = \epsilon) \land (\overline{y} = \overline{y}) \) where \( \overline{y} \) is the vector of all program variables except \( x \).
- If \( n \) is a non-deterministic assignment to \( x \), and \( (n, n') \) is labeled with an expression \( expr \) then \( t \) is defined as the transition created from the assignment \( "x := expr"\).
- If \( n \) is a condition \( B \) (of a “while” or an “if” command) and \( (n, n') \) is labeled \( \text{true} \) then \( t \triangleq B \land (pc = l) \land (pc' = \ell') \land (\overline{x} = \overline{x}) \). If the edge is labeled with \( \text{false} \) then \( t \) is defined in the same way, only using \( \overline{\neg B} \) instead of \( B \).

The labeling function \( L \) is defined so that \( p \in L((l, \sigma)) \) iff \( \sigma \models p \) (recall that every \( p \in AP \) is a boolean expression on program variables).

### 2.3 Parallel Programs

A parallel program is a parallel composition \( P = [P_1 \parallel \ldots \parallel P_n] \) of sequential processes. Each process \( P_i \) is a non-deterministic while program with its own set of local variables. The programming language is enriched with communication statements \( \text{send}(\text{proc}, \epsilon) \) and \( \text{receive}(\text{proc}, x) \), where \( \text{proc} \) is the name of another process, \( x \) is a local variable, and \( \epsilon \) is an expression over local variables. Processes are not allowed to examine variables belonging to other processes or update them. Each process has a control flow graph as described above. The semantics of parallel programs is defined as the parallel composition of the transition systems belonging to the individual processes: \( ts(P) = \| \| ts(CF_i)(P_i) \) \( \| \| ts(CF_i)(P_i) \). The parallel composition of transition systems is defined according to an interleaving model with handshaking communications, which is a model used by languages such as CSP [12] and CCS [20]. Every matching pair of communication transitions \( \text{send}(P_i, \epsilon) \) in \( P_i \) and \( \text{receive}(P_j, x) \) in \( P_j \) is combined into a single transition which updates both program counters, and is enabled only when both program counters are at the respective locations of the \( \text{send} \) and \( \text{receive} \). The effect of such a transition is similar to an assignment “\( x := \epsilon \)”. The set of transitions of \( ts(P) \) is the union of the communication transitions and the local transitions of the processes (transitions that execute statements other than communications).

### 2.4 Specifications

The specifications we handle are of the temporal logic specification language CTL*-X [10]. Given a set of atomic propositions \( AP \), a CTL*-X formula is built from:

- atomic propositions \( p \in AP \),
- the boolean operators \( \lor, \land \) and \( \neg \),
- the temporal operators \( U \) (until), \( G \) (always), and \( F \) (eventually), and
- the path-quantifiers \( \mathbf{A} \) (in all paths) and \( \mathbf{E} \) (there exists a path).

We exclude the \( \mathbf{X} \) (next time) operator. The formal definition of the operators is omitted, and can be found in [10].

### 3 Reducing Sequential Programs

This section shows how to translate a single sequential program into a reduced transition system. This will be done by examining its control flow graph, i.e., by a static analysis of the program, so that we never create the full transition system. The reduction we present preserves CTL*-X specifications.
A reduced transition system $rts(CF_P)$ is created from $CF_P$, having less states and less transitions than $ts(CF_P)$. The idea is to take paths in the control flow graph that cannot influence the specification and “fold” them into a single step. We define a set of breaking points, which are nodes in the control-flow graph, so that all the commands that may influence the specification are considered breaking points.

**Definition 3.3:** [Breaking Points] Given a control-flow graph $CF_P$, the set of breaking points $BP$ is a set of graph nodes s.t. $n \in BP$ iff one of the following holds:

1. $n$ is the initial or terminating program location,
2. $n$ is associated with the program location of an assignment that changes a program variable that appears in $AP$,
3. $n$ is a non-deterministic assignment, or
4. $n$ is the head of a “while” statement.

**Definition 3.4:** A finite simple path $v_1, \ldots, v_k$ in a control-flow graph is called *elementary* if both $v_1$ and $v_k$ are breaking points, and no other node on the way is a breaking point.

The set of breaking points was chosen so that elementary paths have two properties. One is that along any elementary path only the first command might influence the specification. This is why assignments that may influence atomic propositions must be breaking points. The second property is that in a single traversal of an elementary path it is possible to compute its underlying semantics, i.e., under what conditions it will execute and what happens to the values of variables when it is executed. This property requires that non-deterministic assignments will be breaking points and that every loop will contain at least one breaking point.

The reduced transition system $rts(CF_P)$ is defined so that the set of states is $BP \times \Sigma$. Every elementary path in $CF_P$ creates a transition in $rts(CF_P)$. For every such path $\tau$ we compute the reachability condition $R_\tau : \Sigma \rightarrow \{true, false\}$ and the state transformation function $T_\tau : \Sigma \rightarrow \Sigma$.

The definition of the reachability condition and the state transformation function is adapted from the Floyd proof system [11].

The reduced transition system $rts(CF_P)$ is defined so that the set of states is $BP \times \Sigma$. Every elementary path in $CF_P$ creates a transition in $rts(CF_P)$. For every such path $\tau$ we compute the reachability condition $R_\tau : \Sigma \rightarrow \{true, false\}$ and the state transformation function $T_\tau : \Sigma \rightarrow \Sigma$.

The reachability condition $R_\tau(x)$ is a condition on the variables at the beginning of $\tau$ that is true if $\tau$ can be traversed. The state transformation function $T_\tau(x)$ is a function on states that describes what happens to the program variables if the computation indeed traverses $\tau$. Both of these are computed syntactically from the control flow graph, by manipulation of terms, as described below.

**Definition 3.5:** Let $\tau = v_1 \rightarrow \ldots \rightarrow v_m$ be a finite path in $CF_P$. We define $R_\tau^v$ and $T_\tau^v$ to be the corresponding reachability condition and transformation function for the suffix $v_k = \ldots \rightarrow v_m$, by induction on $k$ going from $k = m$ to 1.

**Induction basis:**

$R_\tau^v(x) = true$, $T_\tau^v(x) = x$.

**Induction step:**

$R_\tau^v$ and $T_\tau^v$ are defined according to the command labeling the node $v_k$ for $1 \leq k < m$:

**Skip:**

$R_\tau^v = R_\tau^v + 1$, $T_\tau^v = T_\tau^v + 1$.

**An assignment $y := expr$:**

$R_\tau^v = R_\tau^v + 1[y \leftarrow expr]$, $T_\tau^v = T_\tau^v + 1[y \leftarrow expr]$.  

**A non-deterministic assignment $y := \{expr_1, \ldots, expr_k\}$:**

Let expr$_i$ be the label of the edge traversed by $\tau$, then $R_\tau^v = R_\tau^v + 1[y \leftarrow expr_1]$, $T_\tau^v = T_\tau^v + 1[y \leftarrow expr_1]$.

**Positive test:** ($v_k$ is an “if” or “while” and $v_{k+1}$ is the positive son). Let $B$ be the boolean condition of the command, then $R_\tau^v = R_\tau^v + 1 \land B$, $T_\tau^v = T_\tau^v + 1$.

**Negative test:** ($v_k$ is an “if” or “while” and $v_{k+1}$ is the negative son). Let $B$ be the boolean condition of the command, then $R_\tau^v = R_\tau^v + 1 \land \neg B$, $T_\tau^v = T_\tau^v + 1$.

Finally, $R_\tau = R_\tau^1$ and $T_\tau = T_\tau^1$. Notice that the path $v_1 \rightarrow \ldots \rightarrow v_m$ describes a computation that executes the commands labeling the nodes $v_1, \ldots, v_{m-1}$ but does not execute the command at $v_m$. This is the reason that $R_\tau$ and $T_\tau$ do not depend on the command at $v_m$.

In Figure 1, $R_\tau$ and $T_\tau$ are calculated for a given elementary path (marked by bold edges). In this small example the specification does not refer to either of the variables $x, y$ or $z$ so that the
only breaking points are \( l_0 \) and \( l_4 \). The result \( R_\tau = (1 > y) \) means that when the control is at the beginning of the program, there is a computation that travels along \( \tau \) iff \((1 > y)\). The result \( T_\tau(x, y, z) = (1, y + z, z) \) means that if \( \tau \) is traversed then \( x' = 1, y' = y + z \) and \( z' = z \) where \( x, y, z \) are the variable values at the beginning and \( x', y', z' \) are the values at the end of \( \tau \).

Fig. 1. An example of a calculation of \( R_\tau \) and \( T_\tau \)

We can now define the transition \( t \) created from an elementary path \( \tau = v_1, \ldots, v_m \) to be: \( t \triangleq (pc = l) \land (pc' = l') \land R_\tau(\overline{v}) \land T_\tau(\overline{\nu}, \overline{\nu}) \), where \( l \) is the label of \( v_1 \) and \( l' \) is the label of \( v_m \).

Notice that both the selection of the breaking points and the computation of the reachability conditions and state-transformation functions is done automatically. The user is only required to supply the specification.

**Theorem 3.1** The reduced transition system created in the above manner is equivalent w.r.t \( \text{CTL}^ Asterisk \text{-X} \) to the original transition system associated with the program.

Due to space limitations, we present only a skeleton of the proof for the above theorem.

**Proof:** [Skeleton] From [2] we know that two transition systems are equivalent w.r.t \( \text{CTL}^ Asterisk \text{-X} \) formulas if there is a stuttering equivalence relation \( H \) s.t. if \((s, s') \in H \) then for every run from \( s \) in one system there is a corresponding run from \( s' \) in the other system and vice-versa. Two runs are corresponding if they can be partitioned into blocks (finite series of consecutive states) s.t. every state in the \( i \)th block of one run is in the relation \( H \) with every state in the \( i \)th block of the other.

We define a relation \( H \) between states of \( ts(CFP) \) and \( rts(CFP) \) as follows. Every state in \( ts(CFP) \) is of the form \([l, \sigma] \) where \( l \) is a program location (a node in \( CFP \)) and \( \sigma \in \Sigma \). The states of \( rts(CFP) \) are of the same form, only the locations are all in \( BP \). For every location \( l \in BP \) and \( \sigma \in \Sigma \) we set \(([l, \sigma], [l, \sigma]) \in H \). For every state \( s \) of \( ts(CFP) \) in which \( l \not\in BP \) we look at the possible runs from \( s \). Note that if \( l \) is a boolean condition (of an “if” or “while” command) then \((l, \sigma)\) has only one successor, depending on whether \( \sigma \models B \) or \( \sigma \models B' \). The only states that have more than one successor are states with locations that correspond to a non-deterministic assignment, which is a breaking point. Therefore, if the location of \( s \) is not a breaking point then there is a single run from \( s \) to another state \( s' \) with a location which is a breaking point (without passing any other breaking
points on the way). Since the location of $s'$ is a breaking point, $s'$ is a state of $rts(CF_P)$, and we define that $(s, s') \in H$.

It can easily be shown that the above relation is indeed a stuttering equivalence relation. Figure 2 shows how every state in $rts(CF_P)$ corresponds to a block of states in $ts(CF_P)$. Notice that in every block, the only transition that may influence the specification is the first one. Therefore the first state of a block in $ts(CF_P)$ corresponds to one state $rts(CF_P)$, and all the rest of the states in the block (including the last one) correspond to another state in $rts(CF_P)$.

![Stuttering Bisimulation between CF and RF](image)

**Fig. 2.** Stuttering Bisimulation between $CF_P$ and $RF_P$

## 4 Parallel Programs

The common method for handling parallel programs is to translate each process into a transition system and then create the cross product. As mentioned before, the cross product is created by taking the union of all the local transitions from all the processes, and creating a single transition out of each matching pair of communication transitions. It is well known that the main source for the state-space explosion for parallel programs is the cross product of several processes, since the product must include all possible interleavings of the individual processes. It is therefore desirable to reduce the sizes of the processes to a minimum before attempting to examine their cross product. We propose to reduce each process in a similar fashion to the reduction of sequential programs. The only difference is that we need to handle send and receive commands.

Given a parallel program $P = \|P_1\| \ldots \|P_n\|$ we create the control-flow graph $CF_i$ for each process $P_i$. We then create a reduced transition system $rts(CF_i)$ for each process. The definition of $rts(CF_i)$ is the same as for sequential programs, except for the following addition to the definition of breaking points. A node $n$ will be in $BP$ if

1. $n$ is labeled by a communication statement (send or receive), or is the statement following a communication.

Note that from every node of the control-flow graph which is labeled with a communication command there is a single outgoing edge, pointing to the next statement to be executed. The additional breaking points make sure that any elementary path in $CF_i$ that contains a communication command will not contain any other commands. Therefore each communication command is translated into a transition in the reduced transition system. This isolation of communications enables us to synchronize “send” and “receive” commands without involving any other local operations in the same transition.
The cross product of the transition systems $\text{rts}(CF_1), \ldots, \text{rts}(CF_n)$ is created in the same way as if these were the original transition systems. Notice that since each communication command in the reduced flow graphs of the processes is a transition in itself, it is possible to synchronize them.

**Theorem 4.1** The reduced transition system created in the above manner for a parallel program $P$ is equivalent w.r.t CTL*-X to the original transition system associated with $P$.

A skeleton of a proof for the above theorem can be found in the appendix.

## 5 Integration with verification techniques

Our method for reducing the state-space of programs is carried out according to the syntax of the program, creating a reduced model of the program. Once our method has been applied, one can choose any verification method to be used on the result.

A reduced transition system can be used for *state exploration methods*. These are methods that traverse the state space of the transition system on-the-fly, usually by means of a DFS algorithm. The space consumption of such algorithms is proportional to the maximal simple path from the root (which is the maximal depth of the stack during the search). We notice that when creating the reduced transition system we do not have to create the set of program states. We create only the set of transitions, which are represented by first order formulas. These formulas are used as a next-state function that given a state produces the set of successors of that state. The result of using the reduced transition system instead of the original one is that the maximal simple path (a run that doesn’t go through the same state twice) is shorter and therefore the DFS requires less space. Also, since the reduced transition system has less states overall, and particularly less reachable states, the time needed for the DFS is also reduced.

**Partial order reductions** are methods of reducing the state-space traversed by a state-exploration verification algorithm. These methods are based on the observation that sometimes the specification is not sensitive to the different interleavings of computation sequences belonging to processes running in parallel. In general, a partial order reduction method defines for every state a subset of the enabled transitions that will be traversed. This subset is chosen in a way that ensures that although parts of the state-space will not be visited, the result of the search will not change, i.e., the specification is true in the full model iff it is true in the reduced model. Our method for reducing the program creates a program which is still a parallel composition of processes, only that each transition of a reduced process may represent a series of transitions of the original process. We have therefore restricted in advance the possible interleavings between processes. However, the reduced transition system that our method creates may still include several possible interleavings that can be pruned by any partial order reduction method. There is no need to make changes in our method or in the partial order reduction method. Also, since some partial order methods require an initial DFS of the system in order to perform calculations, it is an advantage that the system that these methods get to work on is smaller than the original.

**Symbolic model checking** [4, 18] is a different approach to verification. It considers finite-state systems (either hardware or software with finite types). The system is modeled by a transition relation describing the possible steps between states. This transition relation is kept by means of Binary Decision Diagrams (BDDs) [3]. The advantage of BDDs is that in most cases the representation of the transition system is significantly smaller than the explicit representation. Another advantage is that they allow for efficient manipulation of sets of states, and symbolic model-checking algorithms make use of this fact. Our method can easily be combined with symbolic methods since a reduced transition system can be compiled into symbolic representation in much the same way the original system would be. In fact, the computation of $R_\ell$ and $T_\ell$ can be done symbolically, to produce BDDs representing them for every reduced edge. Since BDDs are representations of formulas, and operations on formulas (such as conjunctions, disjunctions and quantification) are efficient BDD operations, $R_\ell$ and $T_\ell$ can be created according to the boolean formulas that define them by a series of BDD
operations. The transition relation of the whole system is produced by a boolean combination of the BDDs representing each edge. The size of BDDs is difficult to predict. However, the size depends heavily on the number of bits used to represent each state. By reducing the number of program locations, we are sure to reduce the number of bits needed for the representation of a single state, since we reduce the number of bits used for the program location.

Some verification methods which consider weaker specification languages than CTL* use a notion of fairness to enhance the expressibility of the language [6, 4, 17]. A special condition determines which computations are fair, and only those computations are confronted with the specification. The fairness condition is usually given in terms of a formula in the same specification language or a weaker one, and a computation is fair if this condition is true in infinitely many states along the computation. In order to use such methods with our reduction we have to consider the fairness condition in the same way we consider the specification. The only change to the method presented above is that every assignment that changes a variable on which the fairness condition depends must be considered a breaking point. Assuming that the fairness condition is a CTL*-X formula (or belongs to some subset of CTL*-X) then every state in the original system has an equivalent state in the reduced one (w.r.t the fairness condition), and for every fair computation in the original system the corresponding computation in the reduced system is also fair.

6 Experimental Results

6.1 Using Murphi

We chose as an example an algorithm for finding extrema on a unidirectional ring of processes, presented in [9]. Each process is assigned a (unique) number and together they find the maximum of these numbers. For a program with $n$ processes, the numbers $1, \ldots, n$ are assigned to processes non-deterministically, so that the process that gets the number $n$ is the one with the maximum.

We examined two versions of the algorithm. In the first version, the first step each process performs is to choose a number (that no other process has chosen yet). Verifying this version ensures correctness for all possible assignments of numbers to processes. In the second version of the example each process was assigned a number in advance, specifically process number $i$ got $i$ as its assigned number. The first version is much larger (in terms of the number of states in the transition system) and we examined it only for the case of 3 processes. The second version is smaller, and we examined it for the cases of 3, 4 and 5 processes.

In order to evaluate the amount of reduction achieved by our method we translated the algorithm into a Murphi code. We first took the control flow graph of the program and translated each edge into a Murphi rule (thus creating a transition for each edge). We then created a reduced model by creating a Murphi rule for each elementary path in the control flow graph. We did this for 2 different specifications. The specifications refer to properties of the extrema finding algorithm.

Figure 3 shows the resulting number of states in the original model and in the reduced model for both specifications, for the first version of the example. Figure 4 gives the results for the second version of the example. The tables also give the number of edges in the model, and the time it took Murphi to traverse the graph. The numbers in parenthesis give the relative size of a reduced model w.r.t the full model. The examples were run using a 30M hash table.

The second specification refers to more program variables than the first specification, and therefore creates more breaking points. This is the reason that the reduced models according to the second specification are larger than the reduced models according to the first specification. However, it is easy to see that in all cases the reduced models are significantly smaller than the original model.

7 Directions for Future Research

We expect that there are several ways of making the reduced model even smaller.
The numbers in parenthesis are the percentage of the reduced model w.r.t the full model.

**Fig. 3.** Results of reductions for the full implementation

<table>
<thead>
<tr>
<th>Processes</th>
<th>No. of States</th>
<th>No. of Edges</th>
<th>Time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 Processes</td>
<td>Full Model</td>
<td>4099</td>
<td>16309</td>
</tr>
<tr>
<td>Reduced by Spec. 1</td>
<td>(15%) 608</td>
<td>(12%) 1217</td>
<td>(14%) 0.65</td>
</tr>
<tr>
<td>Reduced by Spec. 2</td>
<td>(19%) 750</td>
<td>(17%) 1750</td>
<td>(19%) 0.86</td>
</tr>
<tr>
<td>4 Processes</td>
<td>Full Model</td>
<td>35069</td>
<td>118424</td>
</tr>
<tr>
<td>Reduced by Spec. 1</td>
<td>(8%) 2680</td>
<td>(6%) 758</td>
<td>(5%) 2.51</td>
</tr>
<tr>
<td>Reduced by Spec. 2</td>
<td>(12%) 4183</td>
<td>(11%) 12512</td>
<td>(9%) 4.29</td>
</tr>
<tr>
<td>5 Processes</td>
<td>Full Model</td>
<td>out of memory</td>
<td>-</td>
</tr>
<tr>
<td>Reduced by Spec. 1</td>
<td>14888</td>
<td>5413</td>
<td>15.14</td>
</tr>
<tr>
<td>Reduced by Spec. 2</td>
<td>21459</td>
<td>105486</td>
<td>39.56</td>
</tr>
</tbody>
</table>

The numbers in parenthesis are the percentage of the reduced model w.r.t the full model.

**Fig. 4.** Results of reductions for the limited implementation

One way of further reducing the size of the transition system is to minimize the number of breaking points. For example, if the specification language is restricted to LTL-X, which is LTL without the $X$ operator, then the non-deterministic assignments do not have to be breaking points. We expect that there are other such optimizations that can be done.

Another possible approach is to perform data-flow analysis on the program to complement the control-flow analysis. Data-flow analysis techniques examine the control-flow graph of the program and compute data-related information. These methods can, for example, compute for every definition of a variable (an assignment into it) the places in the program where the value assigned might be used. It is likely that such information can help in reducing the size of the representation of the program, since we may know in advance that certain combinations of values and program counter locations need not be taken into account.

Finally, we intend to explore the possibility of extending the approach to other communication models such as asynchronous communications or shared variables.

**References**


Appendix

A Proof of correctness for parallel programs

Following is a proof skeleton for theorem 4.1. As before, to prove that our reduction preserves CTL*-X specifications we define a stuttering equivalence relation $H$.

**Proof:**[Skeleton]

We use the same relations $H_i$ between $ts(CF_i)$ and $rts(CF_i)$ that were defined for sequential programs. The relation $H$ is the combination of these "local" equivalence relations, so that $\{(v_1, \ldots, v_n), (r_1, \ldots, r_n)\} \in H \Leftrightarrow \{(v_1, r_1) \in H_1 \land \ldots \land (v_n, r_n) \in H_n\}$.

To show that $H$ is a stuttering equivalence relation we need the following definition:

**Definition A.6:** A transition $t$ in $ts(CF_i)$ is called *distinct* if it corresponds to an edge of $CF_i$ that exits a breaking point.

The set of distinct transitions includes all the transitions that might influence the specification, i.e. change the value of a variable that appears in an atomic proposition. Notice, however, that it also includes transitions that cannot influence the specification, for example - if the breaking point was the head of a "while" statement.

Given a run $\pi = s_1 t_1 s_2 t_2 \ldots$ of $ts(CF)$ and a state $r_1$ of $rts(CF)$ s.t. $(s_1, r_1) \in H$ we show that there exists a corresponding run $\pi'$ of $rts(CF)$ from $r_1$. Every state of $\pi$ is a combination of local states: $s_k = [s_{1k}, \ldots, s_{nk}]$. We mark the distinct transitions in $\pi$ and then divide $\pi$ into blocks so that each block begins after a distinct transition was executed and ends before the next distinct transition is taken (see figure 5).

![Fig. 5. Partitioning of a run into Blocks](image)

**Lemma A.1:** Let $B = s_1, \ldots, s_k$ be a block of $\pi$, and let $r_1$ be a state of $rts(CF)$ s.t. $(s_1, r_1) \in H$. Then for every $2 \leq i \leq k$, $(s_i, r) \in H$.

To see why this is true, notice that for every process $i$, if $(s_i, r_i) \in H_i$ and $P_i$ performs a non-distinct operation, it can lead only to a state which $H_i$ connects to $r_i$.

**Lemma A.2:** Let $s$ and $r$ be states in $ts(CF)$ and $rts(CF)$ respectively s.t. $(s, r) \in H$ and let $t$ be a distinct transition of some process $P_i$ which is enabled in $s$ and leads to a state $s'$. Then there is a corresponding transition $t'$ which is enabled in $r$, and leads to a state $r'$ s.t. $(s', r') \in H$.

If $t$ is not a communication then the difference between $s$ and $s'$ is only in the $i$th element. After $t$ is taken from $s^i$ (into $s'^i$) there is only one possible run in $ts(CF_i)$ to the next state which is at a breaking point. If we take the program locations along this run we can recreate the corresponding
elementary path in \( CF_i \). This path was used to create a transition in \( rts(CF_i) \), which is the transition we choose as \( t' \). The transition \( t' \) leads from \( r' \) to some \( r'' \), and by the definition of \( H_i \) we have that \( (r', r'') \in H_i \).

If \( t \) is a communication then it is also a transition of \( rts(CF) \) and the corresponding states are matched by \( H \).

The combination of the above two lemmas allows us to repeatedly choose transitions in \( rts(CF) \) so that each resulting state corresponds to a block in \( \pi \), to create the desired run \( \pi' \).

For the other direction, we are given a run \( \pi' \) in \( rts(CF) \) and we need to create a corresponding run in \( ts(CF) \). This is simple since every transition in \( rts(CF) \) was created from an elementary path in one of the graphs \( CF_i \), or from a communication. If it is a communication then the same transition exists in \( ts(CF) \) and we take it. Otherwise, we look at the path in \( CF_i \) that created the transition and execute each transition belonging to each edge along the path. The states we pass on the way will create a block which is equivalent to a single state in \( \pi' \).