Static Analysis for State-Space Reductions Preserving Temporal Logics

Karen Yorav and Orna Grumberg

Computer Science Department
The Technion
Haifa 32000, Israel

e-mail: {orna,last}@cs.technion.ac.il

Abstract

In this paper we present two methods that use static analysis of parallel programs to create reduced models for them. Our algorithms examine the control-flow graph of a program (the syntax) and create a 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 temporal logic specifications.

The two methods are orthogonal in their approach. The first, called path reduction, reduces the state-space by compressing computation paths. This method reduces the number of steps each computation takes. The second method, called dead variable reduction, reduces according to the variable domains. It identifies classes of equivalent states which differ only on variable values (and not the program counter) and uses a representative for each class. Our target is not only to be able to reduce according to both approaches, but also to find out which approach is more useful.

Our algorithms are based on syntactic manipulation of expressions, thus enabling us to handle programs with variables over finite as well as infinite domains. Both methods can easily be combined with either explicit state or symbolic methods (and with each other).

We used the Murphi verifier to test the amount of reduction achieved by both methods. We let Murphi perform a DFS search and compared the sizes of the original and reduced transition systems, for several examples and according to both reductions. The results show that path reduction gives significant reductions, while the effects of dead-variable reduction are less impressive. We discuss the differences between the approaches, and the reasons for these results.

1 Introduction

1.1 The state explosion problem

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.

1.2 Static analysis

A program can be described in two levels, a syntactic and a semantic level. A common representation for the syntactic level is a control-flow graph which is a data-structure that represents the syntactic structure of a program. 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 automatic verification, we require a semantic model which represents the set of possible computations of the program. The semantic representation of a program is often 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 sizes of their domains. It is the size of the transition system which creates the difficulty in automatic verification.

Static analysis is the process of examining the control-flow graph of a program (the syntax) to extract information on its semantics, without creating the semantic model. Our purpose is to use static analysis in order to create a smaller semantic model for the program, which we call the reduced transition system of the program. The reduced transition system is built so that it is equivalent to the original transition system of the program, i.e. a given specification is true in the original transition system iff it is true in the reduced one. The specifications we consider are formulas of the logic CTL*, and the logic CTL*-X which is similar to CTL*, but without the next-state operator.

Our algorithms for static analysis are based on syntactic manipulation of expressions, and therefore we allow variables with both finite and infinite domains. When the domains are finite, our methods can be used for automatic verification using an explicit representation of the transition system as well as for verification using a BDD [5] representation. In either case, the verification algorithm itself is not changed, it just receives a smaller model to work on.

Using static analysis we can create a reduced transition system for a program directly out of the control-flow graph, and never build the full transition system. We are therefore 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.

Another important advantage of using static analysis is that in order to implement our reductions, 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 reductions is during the (very short) compilation stage and not in the verification process.

When developing a static analysis method for reducing program models we came up with
two basically orthogonal approaches. We can examine the control of the program, which is the program counter, or we can examine the values that variables can have (the data). Reducing according to control means creating a model that performs as fewer steps in order to achieve the same goal, whereas reducing according to data means creating a model that uses a smaller part of the variable domains.

In this paper we present and compare two methods that use static analysis to create reduced models for programs. The first method, called path reduction, reduces according to control, and the second, called dead-variable reduction, reduces according to data. Both methods automatically create a reduced transition system directly out of the syntax of the program (the control-flow graph), thus avoiding the need to create the full transition system. The reductions are independent of each other and can be used together on the same program. In such a case we perform both analyses on the program, and then create a single transition system that is reduced according to both methods (there is no need to create intermediate models according to one method or the other).

Path reduction creates an equivalent transition system in which there are less program counter locations. In path-reduction, we identify paths in the control flow graph on which a process performs a series of consecutive operations that do not influence the specification. Each such path is replaced by a single transition, representing the computation along this path (instead of a series of transitions).

Dead-variable reduction reduces the transition system by excluding some of the possible values that variables can take at given points in the program. We find places in the program in which the value of a given variable is insignificant, and prune out of the program model all the states that differ only on that variable. A variable $x$ is dead at a certain point in the code if on all computations from that point on a value is assigned to $x$ before its value is read. This means that the current value stored in the dead variable can not influence the computation. This definition is used traditionally in compiler optimization methods. We use this information in order to reduce the state space of the program by ignoring variable values when the variables are dead.

Furthermore, we expand the traditional definition of dead variables so that a variable can be partially dead. Instead of variables being either dead or not at a given point in the program, we define a condition that implies that the variable is dead. Given a variable $x$, we compute a condition $\text{dead}(l)$ for every program location $l$ so that if $\text{dead}(l)$ is true then the value of $x$ at the program location $l$ can be discarded. Otherwise, we keep the value of $x$ as usual. The conditions $\text{dead}(l)$ are computed on the control-flow graph of the program, without building the state-transition graph that represents the program. Once we have calculated these conditions, we can create a smaller state-transition graph representing the same program.

We used Murphi [22] to test the amount of reduction achieved by our methods. 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 several example programs and translated them into Murphi. We then constructed Murphi descriptions of the reduced systems created by our methods. 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 experiments with path reduction gave reduced models that were between 8% to 37% of the original model. This is a significant reduction and implies that path reduction can make
a difference in verification of programs. The results for dead-variable reduction, however, show a very small reduction - the reduced models were between 87% to 100% of the original model. It is possible that these results are influenced by our choice of examples, and there may be examples where dead-variable reduction will be more productive.

1.3 Related works

Our path reduction method is closely related to partial order reductions [24, 26, 25]. 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 path reduction also has the effect of excluding some of the possible interleavings because when we take a series of operations a process P does and meld them into a single transition we exclude all possible interleavings 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. The dead-variable reduction can also be considered a partial order reduction since it excludes some of the successors of a state in which x is dead.

A similar type of reduction to our path reduction was introduced by Miller and Katz in [20]. 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 transition system that is somewhat reduced from the original transition system, 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.

The closest related work to the dead-variable reduction is [2], where a live variable analysis is used to create a reduced model for asynchronous processes that communicate via queues. The analysis they present is similar to what we present as fully dead variable analysis. Our dead-variable reduction is more effective since it allows variables to be partially dead (these terms are explained in the appropriate section). The example used in [2] shows better results than ours. However, we believe that this reduction is achieved because the example was tailored for demonstrating the effectiveness of the method. Our examples are implementations of known protocols.

In [15, 17] static analyses are used for partial order reductions. An analysis of the statements in the program ([15]) or of the control-flow graphs of the processes [17] 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 path reduction does not only inhibit certain interleavings but also “folds”
computation segments into a single step. Another important distinction is that our reduction works for the full CTL* specification language whereas the reductions presented in [15, 17] are used only for the subset LTL-X.

Our work is also related to works like [1, 6, 9, 16] that use abstract interpretation techniques [8] 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.

The paper is organized as follows. Section 2 gives some preliminary definitions which are needed later on. Section 3 presents the reduction according to control-flow paths and Section 4 presents the reduction according to dead variables. Section 5 shows how both reductions can be combined with different verification techniques. Finally, Section 6 gives some results of using our reductions on small examples, and Section 7 gives our conclusions.

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 to "real" programming languages.

\[
P \rightarrow \quad \text{skip} \mid \nonumber
 x := \text{expr} \mid \nonumber
 x := \{\text{expr}_1, \ldots, \text{expr}_n\} \mid \nonumber
 P_1 ; P_2 \mid \nonumber
 \text{if } B \text{ then } P_1 \text{ else } P_2 \text{ fi } \mid \nonumber
 \text{while } B \text{ do } P_1 \text{ od}
\]

where \(x\) is a program variable, \(\text{expr}_i\) are expressions over program variables, and \(B\) is a boolean condition. The statement \(x := \{\text{expr}_1, \ldots, \text{expr}_n\}\) is a non-deterministic assignment, after which \(x\) will contain the value of one of the expressions \(\text{expr}_1, \ldots, \text{expr}_n\). 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. In Section 2.3 we add procedures to this language, and in Section 2.4 we introduce parallel composition of sequential programs.

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 labeled with the program counter location of the statement it represents, and the edges capture the possible transitions between locations.

- A node representing a simple assignment or a "skip" statement has a single out-going edge pointing to the node of the next statement to be executed.
• A node representing a non-deterministic assignment \( x := \{exp_1, \ldots, exp_k\} \) has \( k \) outgoing edges labeled with the expressions \( exp_1 \) through \( exp_k \), 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 (in addition to the program-counter location) and has two outgoing 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 (in addition to the program-counter location) and has two outgoing edges labeled “true” and “false”, pointing, to the body and the next statement after the while respectively.

• We add an extra program-counter location, called the end location, which represents the termination point of the program. The control-flow graph node of the end location is the successor of the last statement of the program, and has no outgoing edges.

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

• \( L : S \to 2^{AP} \) 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 \( \overline{x} = (x_1, \ldots, x_n) \) be the vector of program variables, and let \( pc \) be the variable for the program counter. We create another set of variables \( \overline{x}' = (x'_1, \ldots, x'_n) \) and \( pc' \) to represent the next state. A transition \( t \) will be represented by a formula \( t(pc, \overline{x}, pc', \overline{x}') \) such that every assignment \((l, \sigma, l', \sigma')\) that satisfies it represents a step \((l, \sigma, (l', \sigma')) \in t\). We write \( s \to s' \) when there exists a transition \( t \in Tr \) such that \((s, s') \in t\). 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 \( CF_P = \langle N, E \rangle \) induces a transition system \( ts(CF_P) = \langle N \times \Sigma, I, Tr, L \rangle \) where \( \Sigma \) is the set of all valuations to the program variables (without the program counter). The initial states set is \( I = \{n_0\} \times \Sigma \) 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 associated with \( n \). Let \( l \) be the label (program counter location) of \( n \), and \( l' \) the label of \( n' \). Then:
• If \( n \) is an assignment \( x := e \) then \( t \triangleq (pc = l) \land (pc' = l') \land (x' = e) \land (\vec{y}' = \vec{y}) \) where \( \vec{y} \) 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 \( true \) then \( t \triangleq B \land (pc = l) \land (pc' = l') \land (\vec{x} = \vec{x}') \). If the edge is labeled \( false \) then \( t \) is defined in the same way, only using \( \neg B \) 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 Procedures

We can enrich our language with procedure declarations:

\[
\text{Proc proc\_name(Ref } a_1, \ldots, a_n, \text{ Val } b_1, \ldots, b_m \text{) S EndProc}
\]

The variables \( a_1, \ldots, a_n, b_1, \ldots, b_m \) are called the \textit{formal parameters}. The variables \( a_i \) are passed by reference and the variables \( b_i \) are passed by value. The procedure body \( S \) is itself a non-deterministic while program that may include procedure calls, as long as there are no loops in the call graph. This means that we do not allow recursion of any kind. The syntax for procedure calls is:

\[
\text{call proc\_name}(x_1, \ldots, x_n, y_1, \ldots, y_m);
\]

The variables \( x_1, \ldots, x_n, y_1, \ldots, y_m \) are called the \textit{actual parameters}.

The transition system representing a program that includes procedures is in fact a collection of transition systems, one for the main program and one for each procedure body. The combination of transition systems is defined so that when the computation reaches a procedure call it continues to execute transitions representing the procedure’s body, after replacing the formal parameters with the actual parameters. Note that for most verification purposes there is no need to actually create separate representations of a procedure for each call. We can represent the transition system of the procedure body using expressions (formulas) over the formal parameters, and perform the necessary variable replacements only when needed.

When creating the control-flow graph of a program that uses procedures, we create a separate control-flow graph for each procedure. A node representing a procedure call is labeled by the program-counter location of the call, and has a single outgoing edge to the next statement to be executed after the procedure terminates (similar to assignment statements). The control-flow graph of a procedure body may itself contain nodes that represent calls to other procedures (as long as there is no direct or indirect recursion).

### 2.4 Parallel Programs

A parallel program is a parallel composition \( P = [P_1] \ldots [P_n] \) 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 \textit{send}(proc, e) and \textit{receive}(proc, x), where proc is the name of another process, x is a local variable, and e 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, where nodes representing \textit{send} and \textit{receive} commands have a single outgoing edge pointing to the next command after the communication (similar to skip and assignment commands). The semantics of parallel programs is defined as the parallel composition of the transition systems belonging to the individual processes: \( ts(P) = [ts(CF_1)] \ldots [ts(CF_n)] \). The parallel composition of transition systems is defined according to an interleaving model with hand-shaking communications, which is a model used by languages such as CSP [13] and CCS [21].

Every matching pair of communication transitions \textit{send}(P_i, e) in \( P_j \) and \textit{receive}(P_j, x) in \( P_i \) 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 \textit{send} and \textit{receive}. The effect of such a transition is similar to an assignment “\( x := e \)”. The set of transitions of \( ts(P) \) is the union of the communication transitions and the local transitions of the processes.

\section{2.5 Specifications}

We consider the temporal logic specification language CTL* [11]. This logic is widely used for the specification of reactive systems since it is capable of describing properties of systems and the way they change with time. For a comprehensive discussion of CTL* semantics and its expressive power see [11].

Given a set of atomic propositions \( AP \), a CTL* formula is built from:

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

For our path-reduction presented in section 3 we use a sub-set of CTL* called CTL*-X in which the operator \( X \) is excluded. The language CTL*-X is slightly less powerful, since it cannot specify that something should happen in “exactly \( n \) steps”, but this restriction is meaningless for parallel and distributed applications in which the concept of “the next step” is insignificant.

The formal definition of the operators is omitted, and can be found in [11].

When using CTL* specifications for programs we assume that the atomic propositions in \( AP \) are expressions over program variables. Obviously, the only variables that can influence the satisfaction of a formula are those that appear in (at least) one of these expressions. We call such variables \textit{visible variables}.

\section{3 Path Reduction}

Our first reduction is based on compression of computation paths. We identify computation paths along which each state has a single successor, and which can be compressed into a single step. The specification language preserved by this reduction is CTL*-X.
3.1 Path Reduction for Sequential Programs

In this section we define a reduction which can be applied to any sequential program (process) written in our language of non-deterministic while programs. In section 3.2 we use the reduction for sequential programs to create a similar reduction for parallel programs.

3.1.1 The basic reduction

Given a program $P$ and its control-flow graph $CF_P$, a reduced transition system $rts(CF_P)$ is created directly from $CF_P$. The system $rts(CF_P)$ will have less states and less transitions than the original transition system $ts(CF_P)$ (which was defined as the semantics of $P$), but will be equivalent with respect to CTI*-$X$ formulas.

The idea is to take paths in the control-flow graph that cannot influence the specification and compress 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 3.1: 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 visible variable,
3. $n$ is associated with the program location of a non-deterministic assignment, or
4. $n$ is the head of a “while” statement.

Definition 3.2: 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 statement 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 $CF_P$ induces 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 definitions of these functions are adapted from the Floyd proof system [12]. The reachability condition $R_\tau(\tau)$ is a condition on the variables at the beginning of $\tau$ that is true iff $\tau$ can be traversed. The state transformation function $T_\tau(\tau)$ is a function on states that describes the value of the variables at the end of $\tau$ as a function of their values at the beginning of $\tau$, provided that $\tau$ is indeed traversed. Both of these are computed
syntactically from the control flow graph (i.e., the program text), by manipulation of terms, as described below.

**Definition 3.3:** Let \( \tau = v_1 \rightarrow \ldots \rightarrow v_m \) be a finite path in \( CF_P \). We define \( R^k_\tau \) and \( T^k_\tau \) to be the corresponding reachability condition and transformation function for the suffix \( v_k \rightarrow \ldots \rightarrow v_m \), by induction on \( k \) going from \( k = m \) to 1.

**Induction basis:**
\[
R^m_\tau (\vec{\tau}) = \text{true}, \quad T^m_\tau (\vec{\tau}) = \vec{\tau}.
\]

**Induction step:**
\( R^k_\tau \) and \( T^k_\tau \) are defined according to the command labeling the node \( v_k \) for \( 1 \leq k < m \):

**Skip:**
\[
R^k_\tau = R^{k+1}_\tau, \quad T^k_\tau = T^{k+1}_\tau
\]

**An assignment** \( y := expr \):
\[
R^k_\tau = R^{k+1}_\tau[y \leftarrow expr], \quad T^k_\tau = T^{k+1}_\tau[y \leftarrow expr]
\]

**An array assignment** \( a \left[ expr_1 \right] := expr_2 \):
\[
R^k_\tau = R^{k+1}_\tau[a[expr_1] \leftarrow expr_2] = R^{k+1}_\tau[a \leftarrow (a; expr_1; expr_2)], \quad T^k_\tau = T^{k+1}_\tau[a[expr_1] \leftarrow expr_2] = T^{k+1}_\tau[a \leftarrow (a; expr_1; expr_2)]
\]

**A non-deterministic assignment** \( y := \{expr_1, \ldots, expr_t\} \): Let \( expr_i \) be the label of the edge traversed by \( \tau \), then
\[
R^k_\tau = R^{k+1}_\tau[y \leftarrow expr], \quad T^k_\tau = T^{k+1}_\tau[y \leftarrow expr]
\]

**Positive test:** This case is when \( v_k \) is an “if” or “while” and \( v_{k+1} \) is the positive son. Let \( B \) be the boolean condition of the command. Then
\[
R^k_\tau = R^{k+1}_\tau \land B, \quad T^k_\tau = T^{k+1}_\tau
\]

**Negative test:** This case is when \( v_k \) is an “if” or “while” and \( v_{k+1} \) is the negative son. Let \( B \) be the boolean condition of the command. Then
\[
R^k_\tau = R^{k+1}_\tau \land \neg B, \quad T^k_\tau = T^{k+1}_\tau
\]

Finally, \( R_\tau = R^1_\tau \) and \( T_\tau = T^1_\tau \). 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 \( l_1 \) 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 \( \tau \).

We can now define the transition \( t \) created from an elementary path \( \tau = v_1, \ldots, v_m \) to be:
\[
t = (pc = l) \land (pc' = p) \land R_\tau(\vec{\tau}) \land T_\tau(\vec{\tau}, \vec{\tau}') \land \text{where } l \text{ is the program-counter location associated with } v_1 \text{ and } p \text{ is the program-counter location associated with } v_m.
\]

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.

---

1. The notation \((a \leftarrow (a; expr_1; expr_2))\) means that the array \( a \) is substituted with an array which is identical to \( a \) except that in the \( expr_1 \) cell there is the value \( expr_2 \). See [12] for further details on its necessity and use in Hoare style proof systems.
$P :: l_0: x := \{0, 1, 2\};$
$l_1: \text{if } x > y \text{ then }$
$l_2: y := y + z;$
else
$l_3: x := x + z;$
$l_4: f;$

$C P ::$

$$
\tau = l_0 \rightarrow l_1 \rightarrow l_2 \rightarrow l_4
$$

$R^4_\tau(x, y, z) = \text{true} \quad T^4_\tau(x, y, z) = (x, y, z)$
$R^3_\tau(x, y, z) = \text{true} \quad T^3_\tau(x, y, z) = (x, y, z)[y \leftarrow y + z] = (x, y + z, z)$
$R^2_\tau(x, y, z) = \text{true} \land (x > y) = (x > y) \quad T^2_\tau(x, y, z) = T^3_\tau(x, y, z) = (x, y + z, z)$
$R^1_\tau(x, y, z) = (x > y)[x \leftarrow 1] = (1 > y) \quad T^1_\tau(x, y, z) = (x, y + z, z)[x \leftarrow 1] = (1, y + z, z)$

Figure 1: An example of a calculation of $R_\tau$ and $T_\tau$

**Theorem 3.1** The reduced transition system created in the above manner is equivalent w.r.t CTL*-X to the original transition system associated with the program.

The correctness of theorem 3.1 is an immediate consequence of the definition of breaking points, and the reachability condition and state-transformation function. The proof can be found in the appendix.

### 3.1.2 Procedures

In order to create a reduced transition system for a program that uses procedures, we first create control-flow graphs for the program and for all the procedures that it calls. Given a procedure $proc$, any formal parameter $a$ of type $Ref$ is called visible if there is a call to $proc$ that assigns to $a$ an actual parameter $x$ which is visible. Formal parameters of type $Val$ cannot be visible unless we allow the specification to refer to local variables of procedures. We create a reduced transition system for a procedure body in the same way as for regular sequential programs. The only difference is that we add an extra condition to the definition of the set of breaking points: a node $n$ will be in $BP$ if

5. $n$ is labeled by a procedure call, or is the statement immediately following a procedure call.

This addition isolates the procedure call to make it an elementary path of itself in the control-flow graph of the calling procedure. When we construct the reduced transition system for a program with procedures, instead of creating a single transition out of this elementary path, we use the reduced transition system of the procedure body. This replacement (along
with the necessary changing of formal to actual parameters) can be done only when needed, so as to save space.

**3.2 Path Reduction for 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 state-space explosion in 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 as much as possible before attempting to create 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

6. $n$ 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 out-going 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 $rts(CF_1), \ldots, rts(CF_n)$ is created in the same way as if these were the original transition systems.

**Theorem 3.2** 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.

**4 Dead Variables Reduction**

Our second reduction focuses on reducing variable domains. When creating a model for a program the set of states includes all possible assignments to variables. The dead-variables reduction identifies assignments that induce equivalent computations and reduces the size of the model by choosing a representative of each equivalence class.

Instead of presenting the dead-variable reduction in full, we develop it in steps. We begin by introducing a reduction for sequential programs that utilizes fully dead variables. In the next subsection we improve this reduction by considering partially dead variables. We complete the development of the reduction by showing how to create a reduction for parallel programs as well.
4.1 Fully Dead Variables

We say that a variable $x$ is *used* in a statement if the statement is an assignment and $x$ appears in the expression on the right hand side of the assignment, or if the statement is an “if” or a “while” command and $x$ appears in the condition. We say that $x$ is *defined* in a statement if it is the left hand side of an assignment. Notice that in the statement “$x := x + 1$” $x$ is first used, and then it is defined.

**Definition 4.1:** A program variable $x$ is said to be *dead* at a program location $l$ if on every execution path from $l$, $x$ is defined before it is used.

When a variable is dead at a specific program location its value at that point is insignificant since it will not be used. This means that two states that have that location, and differ only in the value given to $x$, will have identical continuations. To make these states equivalent with respect to CTL* we need to make sure that the value of $x$ does not influence the truth of atomic propositions. These conditions are summarized in theorem 4.1 below.

**Definition 4.2:** Let $\Sigma$ be the set of all assignments to program variables, and let $\sigma, \sigma' \in \Sigma$ be two such assignments. We write $\sigma \equiv_{-x} \sigma'$ if $\sigma(y) = \sigma'(y)$ for every program variable $y$ such that $y \neq x$.

**Theorem 4.1** Let $l$ be a program location in $S$, and $x$ a non-visible program variable which is dead at $l$. For any two states $(l, \sigma_1), (l, \sigma_2)$ s.t. $\sigma_1 \equiv_{-x} \sigma_2$ it holds that $(l, \sigma_1) \equiv_{\text{CTL}*} (l, \sigma_2)$.

The proof for this theorem can be found in the appendix.

We can now build a reduced equivalent model for a program, in which we keep only one representative of each equivalence class.

**Definition 4.3:** Let $d$ be a representative value from the domain of $x$. Given a program $P$, the reduced model of $P$ is the transition system $\text{rts}(P)$. Recall that a transition system is created from the control-flow graph of $P$ by defining a transition representing every edge in the control-flow graph. For every edge $n \rightarrow n'$ such that $x$ is dead in $n'$ we create a transition that simultaneously performs the statement in $n$ and an assignment of $d$ into $x$. Let $f$ be the boolean formula representing the transition induced by the edge $n \rightarrow n'$ (as defined in section 2). The reduced transition created here is $f \land (x' = d)$. All other edges are translated into transitions in the usual manner.

As an example, if the statement at location $l$ is $y := \epsilon$, the transition created for this location will simultaneously assign $\epsilon$ into $y$ and $d$ into $x$.

The reduced transition system $\text{rts}(P)$ can be created statically (from the control flow graph of $P$) without building the structure $ts(P)$. The reduced structure will have less reachable states, since every equivalence class from $H_k$ will be represented by a single state, the one that gives $x$ the chosen value $d$. Calculating the locations in which $x$ is dead can also be done statically and efficiently, by examining the text of $P$. Furthermore, in order to produce a smaller model we may perform this reduction for more than one variable. For every variable we wish to use we compute the locations in which it is dead. The definition of the reduced transition system is updated accordingly.

4.2 Partially Dead Variables

We wish to make our reduction more effective (i.e. create an even smaller reachable state-space) by taking into account more information about the possible uses of variables. We
\begin{verbatim}
l_1: if (y < 0) then
   l_2: y := x;
else
   l_3: y := 0;
fi;
l_4: x := 0;
\end{verbatim}

Figure 2: An example of a partially dead variable

notice that in some cases, even though a variable $x$ is not dead at a location $l$, there are possible computations from $l$ on which $x$ will not be used. For example, in figure 2 we see that when control is at location $l_1$ the variable $x$ is used before it is defined only if $y < 0$. For every state $(l_1, \sigma)$ such that $\sigma \not\models (y < 0)$ we can be sure that on every computation that starts from $(l_1, \sigma)$, $x$ is defined before it is used. However, according to the definition of the previous subsection, $x$ is not dead at $l_1$ and therefore there will be no reduction. In this subsection we show how to find such cases, and use this information.

We change our definition of "dead variables". Instead of looking at variables that are dead at a given program location, we look at variables being dead at a given state. Therefore, for a given program location we will have a condition that tells us when $x$ is dead. The method in the previous section can be viewed as a version of this new method, using only conditions true or false.

**Definition 4.4:** Let $x$ be a program variable, $l$ a program location, and $\sigma$ an assignment to the program variables. We say that $x$ is dead at the state $s = (l, \sigma)$ if on all possible runs from $s$ the value of $x$ is not used before it is defined (either $x$ will not be used, or it will be defined before the first time it is used).

This definition is similar to the definition of $x$ being dead at a program location, except that wherever we referred to a program location we now refer to a specific combination of program location and variable values. Again we find an equivalence between states that differ only on a dead variable.

**Theorem 4.2** If a non-visible variable $x$ is dead at $(l, \sigma)$, and if $\sigma \equiv_{\leq} \sigma'$, then $(l, \sigma) \equiv_{\text{CTL}} (l, \sigma')$.

We omit the proof of this theorem, but note that it is easy to develop it from the proof of Theorem 4.1.

For the remainder of this section we assume that $x$ is the variable according to which we want to perform our reduction.

We calculate for each program location $l$ a boolean condition over the program variables, called dead($l$), so that for every assignment $\sigma$ it holds that if $\sigma \models \text{dead}(l)$ then $x$ is dead at $(l, \sigma)$. The condition we calculate is an under-approximation since the implication in the other direction might not be true (i.e. it is possible that $x$ is dead at $(l, \sigma)$ and yet $\sigma \not\models \text{dead}(l)$). We compute an under-approximation because calculating the exact condition cannot be done in a single traversal of the control-flow graph.
For the simplicity of our presentation we restrict ourselves to handling only non-array variables, i.e. the variable $x$ for which we want to define the condition $\text{dead}(l)$ is not an array.

We need to calculate $\text{dead}(l)$ for every program location, and we do it by traversing the control-flow graph of $P$, bottom up. At each step, when we calculate the condition $\text{dead}(l)$ for a sub-program, we have already calculated the conditions for its end location.

The first step is to assign a condition to the final program location $\text{end}$: $\text{dead}\left(\text{end}\right) = \text{true}$. We now describe how to calculate the condition for a sub-program, given that we have already calculated it for its end location.

- For the sub-program $l$: skip $l'$:
  \[\text{dead}(l) = \text{dead}(l').\]

- For the sub-program $l$: $x := exp$ $l'$:
  If the expression $exp$ does not use $x$ then $\text{dead}(l) = \text{true}$. Otherwise, $\text{dead}(l) = \text{false}$.

- For the sub-program $l$: $y := exp$ $l'$ ($y \neq x$):
  If the expression $exp$ does not use $x$ then we change the condition according to the assignment: $\text{dead}(l) = \text{dead}(l')[y \leftarrow \text{exp}]$. If $exp$ uses $x$ then $\text{dead}(l) = \text{false}$.

- For the sub-program $l$: $y := \{exp_1, \ldots, exp_n\}$ $l'$:
  We add up the influences of all the possible assignments. If $x$ is used by (at least) one of the expressions $exp_1, \ldots, exp_n$ then $\text{dead}(l) = \text{false}$. Otherwise, if $y = x$ then $\text{dead}(l) = \text{false}$. If $x$ is neither used nor defined by the assignment then $\text{dead}(l) = \Lambda_{j=1,\ldots,n} \text{dead}(l')[y \leftarrow \text{exp}_j]$.

- For the sub-program $l$: $a[exp_1] := exp_2$ $l'$:
  If either $exp_1$ or $exp_2$ uses $x$ then $\text{dead}(l) = \text{false}$. Otherwise $\text{dead}(l) = \text{dead}(l')[a[exp_1] \leftarrow exp_2] = \text{def}(l')[a \leftarrow (a; exp_1: exp_2)]$.

- For the sub-program $l$: if $B$ then $l_1$: $S_1$ else $l_2$: $S_2$ fi $l$:
  We use a recursive call to calculate the conditions $\text{dead}(l_1)$ and $\text{dead}(l_2)$, using $\text{dead}(l_e)$ as input for both calculations. If the condition $B$ does not use $x$ we can set $\text{dead}(l) = (B \land \text{dead}(l_1)) \lor (\neg B \land \text{dead}(l_2))$. If, on the other hand, the condition $B$ uses $x$ then $\text{dead}(l) = \text{false}$.

- For the sub-program $l$: while $B$ do $l_1$: $S_1$ $l_1'$ odd: $l_2$:
  Similarly to the “if” case, if $B$ uses $x$ then $\text{dead}(l) = \text{false}$.

  Otherwise, we use a recursive call to calculate $\text{dead}(l_1)$. The input to this call (a value for $\text{dead}(l_1')$) is the “safest” approximation we can give, since we do not have any information on what happens at the end of the body after each iteration. If $x$ does not appear in $S_1$ at all, which can be checked while parsing the program, and if $\text{dead}(l_e) = \text{true}$ then we assume $\text{dead}(l_1') = \text{true}$. Otherwise, we have to assume $\text{dead}(l_1') = \text{false}$.

  When the recursive call for $S_1$ is done we define: $\text{dead}(l) = (B \land \text{dead}(l_1)) \lor (\neg B \land \text{dead}(l_e))$.

15
The above definition can be further optimized by adding traversals through the body of each loop. The important characteristic that we must maintain is that we traverse the control-flow graph of the program a constant number of times, and therefore it is more efficient than model checking on the full model of the program. Notice that the reason we need to perform approximations is the while loop. All other constructs create an exact computation of $\text{dead}$.

One optimization, which we used in our examples, is to traverse each loop twice so that we can identify situations in which $x$ is dead at the top of the body (i.e. at location $l_1$), although it is used somewhere inside. Assume that $\text{dead}(l_1) = \text{true}$. In order to have $\text{dead}(l_1) = \text{true}$ according to the above algorithm $x$ must be defined on every path from the beginning to the end of the loop body. Instead, we propose to first compute $\text{dead}$ on $S_1$ under the assumptions $\text{dead}(l_1') = \text{true}$. If under this condition we find that $\text{dead}(l_1) = \text{true}$ then we can conclude that $x$ can never be used before it is defined inside the loop. We can therefore set $\text{dead}(l) = \text{true}$. If, on the other hand, the calculation result is that $\text{dead}(l_1) = \text{false}$ then we have calculated our condition for $S_1$ under false assumptions, and therefore cannot use these results. We then do another round on $S_1$, this time with the original safe assumption: $\text{dead}(l_1) = \text{false}$.

Notice that the condition $\text{dead}(l)$ can never be dependent on $x$. When it depends on a variable $y$ it is because during its calculation we passed a statement that evaluates an expression involving $y$. However, a statement that evaluates an expression that depends on $x$ is a use of $x$, after which we have $\text{dead}(l) = \text{false}$.

We can now define a reduced transition system according to (partially) dead-variables reduction:

**Definition 4.5**: Given a program $P$ and a choice of a non-visible variable $x$, we compute $\text{dead}(l)$ for all the locations in $P$. We redefine $\text{rts}(P)$ so that every edge $n \rightarrow n'$ in the control-flow graph is translated into a reduced transition according to the information we calculated. Let $f$ be the function describing the transition associated with $n$ but not defining the next-state value for $x$ ($x'$), and let $l$ be the program counter location of $n$. If $x$ is used by the statement at $l$, then there is no change in the definition of this transition. Otherwise, we define the reduced transition to be: $f \wedge (\text{dead}(l') \land (x' = d)) \lor (\text{dead}(l') \land (x' = x))$.

The result is that all the states that were excluded, and perhaps all of their descendants, will not be traversed. In effect - we have pruned parts of the state-space. To show that the pruned structure is CTL* equivalent to the original structure we build a new bisimulation relation.

**Definition 4.6**: $H = H_1 \cup H_2 \subseteq S \times S$ such that:

$H_1 = \{((l, \sigma_1), (l, \sigma_2)) \mid \sigma_1 \models \text{dead}(l) \text{ and } \sigma_1 \equiv_{\neg x} \sigma_2\}$

$H_2 = \{((l, \sigma), (l, \sigma)) \mid \sigma \not\models \text{dead}(l)\}$

It is easy to see that this is truly a bisimulation relation.

From here on, when referring to dead-variables reduction, we are referring to the reduction according to partially dead-variables.

### 4.2.1 Procedures

For completeness reasons it is important to note that there is no problem with handling procedures when performing dead-variables reduction. The straightforward solution would...
be to replace each call to a procedure by the procedure body (with the necessary exchange of actual parameters instead of formal parameters) resulting in a single sequential program. There are several ways, however, in which procedures can be handled in a more efficient manner.

Any procedure can be handled in the same way that we treat the body of a while loop. We make safe assumptions on the ending point of the procedure, and get safe results for the entrance to the procedure. This means that we might not get accurate results, but we can spare computation time since we analyze each procedure body only once.

If we require the most accurate results possible, there are still some optimizations that can be made. We notice that at the end of a procedure all the local variables are dead. The computation of $\text{dead}(l)$ for a local variable $x$ of a procedure can be done once (instead of once for each procedure call). Variables that are passed by value cannot be defined by the procedure and are very likely to be used, so it is safe to assume that they are (totally) live at the entrance to the procedure (i.e. $\text{dead}(l) = \text{false}$) without checking individually for each call.

### 4.3 Dead-Variable Reduction for Parallel Programs

The dead-variable reduction for parallel programs proceeds in a similar manner to the path reduction. We first reduce each process separately and then create the cross product of the reduced models. However, in order to perform the dead-variable reduction on a single process we have to augment the computation of $\text{dead}(l)$ with instructions for handling communication commands:

- For the sub-program $k$: send($P_i, \text{exp})^l$:
  
  If $\text{exp}$ uses $x$ then $\text{dead}(l) = \text{false}$. Otherwise, $\text{dead}(l) = \text{dead}(l')$.

- For the sub-program $k$: receive($P_i, x$)$^l$:
  
  $\text{dead}(l) = \text{true}$.

- For the sub-program $k$: receive($P_i, y$)$^l$ ($y \neq x$):
  
  Since any value may be assigned to $y$ by this operation, to be sure that $\text{dead}(l')$ is true after executing the command we must require that it will be true for all possible values assigned to $y$: $\text{dead}(l) = \forall y. \text{dead}(l')$.

This allows us to compute the condition $\text{dead}(l)$ for every location of a process and for each process separately.

**Theorem 4.3** Let $P = [P_1 \parallel \ldots \parallel P_n]$ be a parallel program. Then the parallel composition of the dead-variable reduced structures for the processes is bisimilar to the parallel composition of the original structures. Formally, if $ts(P) = ts(P_1) \parallel \ldots \parallel ts(P_n)$ and $rts(P) = rts(P_1) \parallel \ldots \parallel rts(P_n)$, then $ts(P) \equiv_{\text{CTL}} rts(P)$.

**Proof:** Let $H_i$ be the bisimulation relation from definition 4.6 between $ts(P_i)$ and $rts(P_i)$. Every state $s$ in $ts(P)$ is a tuple $(s_1, \ldots, s_n)$ such that $s_i$ is a state of $ts(P_i)$, and similarly every state $t$ in $rts(P)$ is a tuple $(t_1, \ldots, t_n)$. We define a relation $H$ such that $(s, t) \in H$ iff for every $i$ $(s_i, t_i) \in H_i$. It is easy to see that since every $H_i$ is a bisimulation relation, $H$ is also a bisimulation relation. □
5 Integration with verification techniques

Our methods for reducing the state-space of programs are carried out according to the syntax of the program, creating a reduced model of the program. Given a program $P$ it is possible to apply one or both of the reductions. Furthermore, when applying the dead-variable reduction we can do so for more than one variable. Once our methods have been applied, one can choose any verification method to be used on the result. We now describe in more detail how our reduction methods can be incorporated into several well known verification techniques.

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. Examples of verification tools that use such methods are Murphi [22] and SPIN [14]. 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). The time complexity is linear in the number of reachable states. We notice that when creating a 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. In the case of path-reduction the maximal simple path (a run that does not go through the same state twice) is shorter and therefore the DFS requires less space. In both reductions the reduced transition systems have less reachable states and so the time needed for the DFS is also reduced.

Partial order reductions [24, 26, 25] are methods of reducing the state-space traversed by a state-exploration verification algorithm, and are implemented in state-space exploration tools such as SPIN. 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. The path reduction method presented in section 3 can be compared to partial order reduction methods. Path reduction 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. Therefore some of the possible interleavings between processes are restricted beforehand. This reduction cannot be achieved by partial order reduction methods, but the reduced transition system that path reduction creates may still include several possible interleavings that can be pruned by a partial order reduction method. There is no need to make changes in our method or in the partial order reduction method in order to combine the two. Note also that 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 one.

Symbolic model checking [5, 19] 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) [4]. 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 methods can easily be combined with symbolic methods since a reduced transition system can be translated directly into a symbolic representation in the same way the original system would be. In fact, the computation of $R_T$ and $T_T$ for path reduction and $used$ and $def$ for dead-variable reduction can be done symbolically, to produce a BDD representation of the resulting system. Since BDDs are representations of formulas, and operations on formulas (such as conjunctions, disjunctions and quantification) are efficient BDD operations, $R_T$, $T_T$, $used$ and $def$ 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, it depends heavily on the number of bits used to represent each state. Path reduction reduces the number of program counter locations, and therefore reduces the number of bits needed for the representation of a single state. For dead-variable reduction it might be preferable to replace the assignment of a single chosen value $d$ into $x$ with the a non-deterministic assignment that allows $x$ to have any value. In this way, $x$ might be quantified out of the transition relation (or parts of it). It also appears that in some cases it would be better to use the fully-dead version of reduction (subsection 4.1) so as not to introduce dependencies between variables.

Some verification methods which consider weaker specification languages than CTL* use a notion of fairness to enhance the expressibility of the language [7, 5, 18]. 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 is that variables that appear in the fairness condition will also be considered visible. Assuming that the fairness condition is in the language that the reduction preserves (CTL* or CTL*-X), 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

In order to evaluate our reductions we chose several examples and translated them into Murphi code. The Murphi language [22] consists of a list of rules, where each rule has a guarding condition and a body. A rule is executable if its guard evaluates to true. The semantics of Murphi programs consists of a loop in which the set of executable rules is computed, one executable rule is chosen non-deterministically and then its body is executed. This process continues until (if ever) there are no executable rules. We used Murphi to perform a traversal of the state-space of each example, at the end of which we get the number of states and edges in the reachable state-space of the model. Each example was manually translated into a Murphi program that represents its semantics ($ts(P)$). We then performed our reductions using the original program text (the control-flow graph) and created a new Murphi program that represents the reduced model for that example ($rt\,ts(P)$). For each
example we performed path reduction, dead-variable reduction, and then a combination of both reductions. All the examples were run with a 350M hash-table. The table in figure 3 summarizes the results we obtained using our reductions. For each example we give the number of states and edges in the model, and the time it took Murphi to complete the traversal. Lines in the table that say 'failed' signify examples in which the hash-table was filled before the whole structure was traversed. The numbers in parenthesis show the relative size of the reduced model with respect to the non-reduced model. In figure 4 we give a block diagram summarising the results for which the non-reduced example did not fail. The reduced models are given as a percentage of the non-reduced model.

The first example, *slide*, is a program that simulates the sliding window communication protocol between a sender and a receiver. The results in the table were obtained for an example in which both the receiver and the sender windows were of length 2. The variables chosen for the dead-variable reduction were temporary variables used to store incoming messages. The path-reduction was done with respect to a specification that states that the sender does not advance its window before the receiver has received the first message in the sender's window.

The second example, called *linked-list*, is an example of a sorting algorithm that uses a distributed linked-list of processes. It consists of several processes connected in a row so that each of them keeps a number. When a process receives a new number from the process on its right, it compares it with the number it already has. It keeps the larger of the two and sends the smaller to the next process in the line. The example includes also a main process that inputs a list of numbers and sends them to the first in line (input is simulated by non-deterministic assignment). The dead-variable reduction was done with respect to a temporary variable that each node keeps whenever it holds two numbers (before it sends one of them to the next node in the line). The path reduction used a specification that states that whenever a node is expecting a number from the preceding node, the preceding node is about to send a number. We ran this example using 4 and 5 nodes in the list. For the case of 5 nodes the full model, and the dead-variable reduced model, were too large to handle. However, after performing path reduction the resulting reduced model was small enough for the traversal to terminate, and combining both reductions resulted in an even smaller model.

The third example, *find-max*, is an algorithm for finding extrema on a unidirectional ring of processes, presented in [10]. 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. The variables used for the dead-variable reduction were temporary variables used to hold incoming messages. For this example we created path-reduced models according to two different specifications. The algorithm works in phases, and during each phase each process may either be active or not. In the last phase there is only one active process, and it holds the maximum number. The first specification makes sure that when a process receives a message from phase \( i \), the sending process is also in phase \( i \). The second specification states that in any given phase, no two active processes hold the same number.

The last example *simple*, is taken from [2]. It involves two processes where the first process repeatedly sends request or switch messages and the second receives them. Each request message is accompanied by a number, chosen non-deterministically. The second process has a state in which a request message results in outputting the number received,
and another state in which the number is ignored. Every time the second process receives a 
\textit{switch} message it switches between these two states. The example is given in the form of 
two state machines, the first having two states and the second three. We translated it into 
two processes in our programming language, with the state kept in a program variable. This 
influenced more states in the transition system of the example because of the introduction 
of a program counter. Obviously, when the number sent with a \textit{request} message is going to 
be ignored, the variable that holds this number is dead. Since the program does nothing else 
than receive the number and output it, the size of the domain of this variable is the main 
factor in the size of the transition system, and hence the significant effect of dead-variable 
reduction. This is also the reason why, in this example, path-reduction is not so effective as 
in other examples.

7 Conclusions

We now discuss our results, first the results for each reduction by itself and then the combi-
nation of the two.

It is clear that dead-variable reduction in itself does not produce significant reduction. 
This can be a result of the nature of our examples. We expect this reduction to be useful 
on programs that perform several tasks, where one task requires many variables that are 
not needed for other tasks. This situation can occur, for example, when the program does 
some numerical computation and then goes on to use only the result of the computation. 
The variables that were used during the computation are now dead. Our examples, being 
hand-produced demonstrative examples, are relatively simple, and perform no intermediate 
computation. It remains to be seen whether real-life examples exhibit similar behavior or 
not.

Path reduction, in contrast, gives significant reduction in the size of the models, and 
as a result also reduces computation times. Using path reduction we get reduced models 
which are between 8% to 37% of the original model. The explanation of this is that a 
sequential program (or process) performs only one operation at each step - either evaluating 
a condition, or assigning into a single variable. By condensing all the operations a program 
does between two observable points into a single step, we create a much more concise model. 
In the last example, we used two different specifications to create the path-reduced model. 
The second specification refers to more variables than the first and therefore creates more 
breaking points. This explains why the reduced model according to the first specification is 
smaller than the reduced model according to the second specification. Notice that in two 
cases (linked-list with 5 nodes and find-max with 5 nodes) we could not traverse the non-
reduced model since it was too large. However, after using path-reduction we got a model 
which we could traverse in full. In these cases we cannot tell the ratio of reduction, since 
we do not know the size of the non-reduced model, but we see that it is significant enough 
to allow us to perform model-checking on programs that are otherwise too large to handle. 
In the case of the find-max example, we could only handle 4 processes in the non-reduced 
version, whereas using path-reduction we successfully completed the traversal of the model 
with 8 processes.

When combining both reductions we get an even smaller model in all of the examples. 
Even though the added efficiency due to dead-variable reduction is not large, it seems useful
<table>
<thead>
<tr>
<th>Example</th>
<th>Reduction used</th>
<th>No. of States</th>
<th>No. of Edges</th>
<th>Time (h:m:s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>slide</td>
<td></td>
<td>2895333</td>
<td>6158396</td>
<td>0:6:35</td>
</tr>
<tr>
<td></td>
<td>dead-vars</td>
<td>(98.6%) 2854863</td>
<td>(98.8%) 6085068</td>
<td>(84.3%) 0:5:33</td>
</tr>
<tr>
<td></td>
<td>path</td>
<td>(36.6%) 1061947</td>
<td>(37.3%) 2302118</td>
<td>(23.5%) 0:1:33</td>
</tr>
<tr>
<td></td>
<td>both</td>
<td>(32.1%) 929967</td>
<td>(33.3%) 2051234</td>
<td>(15.9%) 0:1:23</td>
</tr>
<tr>
<td>linked-list</td>
<td></td>
<td>1203536</td>
<td>4147621</td>
<td>0:13:28</td>
</tr>
<tr>
<td>4 nodes</td>
<td>dead-vars</td>
<td>(87.6%) 1054448</td>
<td>(88.7%) 3679149</td>
<td>(86%) 0:11:35</td>
</tr>
<tr>
<td></td>
<td>path</td>
<td>(37.1%) 446849</td>
<td>(35.3%) 1466866</td>
<td>(21.4%) 0:2:53</td>
</tr>
<tr>
<td></td>
<td>both</td>
<td>(32.6%) 392849</td>
<td>(31.4%) 1304450</td>
<td>(19.9%) 0:2:41</td>
</tr>
<tr>
<td></td>
<td>dead-vars</td>
<td>failed</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>path</td>
<td>12463025</td>
<td>47998708</td>
<td>1:55:55</td>
</tr>
<tr>
<td></td>
<td>both</td>
<td>10213613</td>
<td>39767184</td>
<td>1:38:06</td>
</tr>
<tr>
<td>find-max</td>
<td></td>
<td>7450357</td>
<td>25373736</td>
<td>1:38:47</td>
</tr>
<tr>
<td>4 procs</td>
<td>dead-vars</td>
<td>(100%) 7450357</td>
<td>(100%) 25373736</td>
<td>(100%) 1:38:47</td>
</tr>
<tr>
<td></td>
<td>path (spec1)</td>
<td>(8%) 598913</td>
<td>(6.9%) 1774912</td>
<td>(4.3%) 0:4:15</td>
</tr>
<tr>
<td></td>
<td>path (spec2)</td>
<td>(15.6%) 1168793</td>
<td>(14.4%) 3659832</td>
<td>(11.3%) 0:11:10</td>
</tr>
<tr>
<td></td>
<td>failed</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>path (spec1)</td>
<td>534532</td>
<td>1979482</td>
<td>0:5:41</td>
</tr>
<tr>
<td></td>
<td>path (spec2)</td>
<td>1280152</td>
<td>5040509</td>
<td>0:15:58</td>
</tr>
<tr>
<td></td>
<td>path (spec1)</td>
<td>2517233</td>
<td>11205360</td>
<td>0:39:38</td>
</tr>
<tr>
<td></td>
<td>path (spec2)</td>
<td>-4560483</td>
<td>20309124</td>
<td>1:24:23</td>
</tr>
<tr>
<td>5 procs</td>
<td>path (spec1)</td>
<td>1280152</td>
<td>5040509</td>
<td>0:15:58</td>
</tr>
<tr>
<td></td>
<td>path (spec2)</td>
<td>2517233</td>
<td>11205360</td>
<td>0:39:38</td>
</tr>
<tr>
<td></td>
<td>path (spec1)</td>
<td>-4560483</td>
<td>20309124</td>
<td>1:24:23</td>
</tr>
<tr>
<td></td>
<td>path (spec2)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>simple</td>
<td>dead-vars</td>
<td>(72.3%) 1641400</td>
<td>(74.8%) 3570746</td>
<td>(75.9%) 0:1:19</td>
</tr>
<tr>
<td></td>
<td>path</td>
<td>(67.1%) 1524686</td>
<td>(74.8%) 3566299</td>
<td>(66.3%) 0:1:09</td>
</tr>
<tr>
<td></td>
<td>both</td>
<td>(47.8%) 1085746</td>
<td>(54.7%) 2608879</td>
<td>(50%) 0:0:52</td>
</tr>
</tbody>
</table>

Figure 3: Results for both reductions
to use since it is practically ‘for free’.

References


A Proof of correctness for path reduction of a single process

Following is a proof for theorem 3.1:

The reduced transition system created by path-reduction is equivalent w.r.t CTL* to the original transition system associated with the program.

Proof: [Skeleton] From [3] we know that two transition systems are equivalent w.r.t CTL* 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 $ts(CF_P)$ and $rts(CF_P)$ as follows. Every state in $ts(CF_P)$ is of the form $(l, \sigma)$ where $l$ is a program location (a node in $CF_P$) and $\sigma \in \Sigma$. The states of $rts(CF_P)$ 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 $ts(CF_P)$ in which $l \notin BP$ we look at the possible runs from $s$. We observe that 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. 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 $\sigma \not\models B$. 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 5 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 $rts(CF_P)$.

\[\]

![Diagram](image.png)

Figure 5: Stuttering Bisimulation between $ts(P)$ and $rts(P)$
B Proof of correctness of path-reduction for parallel programs

Following is a proof skeleton for theorem 3.2:
The reduced transition system created by path-reduction for a parallel program $P$ is equivalent w.r.t $CTL^*-X$ to the original transition system associated with $P$.

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 $((v_1, \ldots, v_n), (r_1, \ldots, r_n)) \in H \iff [(v_1, r_1) \in H_1 \land \ldots \land (v_n, r_n) \in H_n]$.

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

**Definition B.1:** 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, when the breaking point was the head of a "while" statement.

Given a run $\pi = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \ldots$ 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: $s_k = (s_{k1}^1, \ldots, s_{kn}^k)$. 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 6)

![Figure 6: Partitioning of a run into Blocks](image)

**Lemma B.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 $2 \leq i \leq k$, $(s_i, r_1) \in H$.

To see why this is true, notice that for every process $i$, if $(s_i, r_1) \in H_i$ and $P_i$ performs a non-distinct operation, it can lead only to a state which $H_i$ connects to $r_i$.

**Lemma B.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 \( s' \) (into \( s'' \)) there is only one possible run in \( t s(CF_i) \) 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 \( r ts(CF_i) \), 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 \( (s'', r'') \in H_i \).

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

The combination of the above two lemmas allows us to repeatedly choose transitions in \( r ts(CF) \) 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 \( r ts(CF) \) and we need to create a corresponding run in \( ts(CF) \). This is simple since every transition in \( r ts(CF) \) 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 \( \pi' \). □

C Proof of correctness for fully-dead variables reduction

Following is the proof for Theorem 4.1: Let \( l \) be a program location in \( S \), and \( x \) a non-visible program variable which is dead at \( l \). For any two states \( (l, \sigma_1), (l, \sigma_2) \) s.t. \( \sigma_1 \equiv_x \sigma_2 \) it holds that \( (l, \sigma_1) \equiv_{\text{CTL}_*} (l, \sigma_2) \).

To prove this theorem we use the following Lemma:

**Lemma C.1:** Let \( x \) be a variable and \( l \) a location such that \( x \) is neither used nor defined by the command at \( l \). Let \( s_1 = (l, \sigma_1) \) and \( s_2 = (l, \sigma_2) \) be two states such that \( \sigma_1 \equiv_x \sigma_2 \). Then for every state \( (l', \sigma'_1) \) such that \( (l, \sigma_1) \rightarrow (l', \sigma'_1) \) there exists a state \( (l', \sigma'_2) \) such that \( (l, \sigma_2) \rightarrow (l, \sigma'_1) \) and \( \sigma'_1 \equiv_x \sigma'_2 \).

**Proof:** Since the command at \( l \) does not use or define \( x \) we know that it is either an assignment which does not involve \( x \) (on either side) or a branching command (“if” or “while”) for which the condition does not involve \( x \). The proof is according to this command.

- Assume that the command at location \( l \) is an assignment \( "y := e" \). All states with location \( l \) have exactly one successor state, with the same location \( l' \). Let \( (l', \sigma'_1) \) be the successor of \( (l, \sigma_1) \) and let \( (l', \sigma'_2) \) be the successor of \( (l, \sigma_2) \). Since \( x \) does not appear in the assignment we know that \( y \neq x \) and that \( \sigma_1(e) = \sigma_2(e) \). Therefore, we can deduce that \( \sigma'_1 \equiv_x \sigma'_2 \).

- Assume that the command at location \( l \) is a non-deterministic assignment \( "y := \{e_1, \ldots, e_n\}" \). Each successor state \( (l', \sigma'_1) \) of \( (l, \sigma_1) \) is a result of assigning one of the expressions \( e_i \) to \( y \). Since \( x \) does not appear in any of these expressions, for every successor \( (l', \sigma'_1) \) of \( (l, \sigma_1) \) there must be a successor \( (l', \sigma'_2) \) of \( (l, \sigma_2) \) which is the result
of assigning the same \( e_i \) to \( y \). Because \( \sigma_1(e_i) = \sigma_2(e_i) \) we conclude that the same value is assigned into \( y \) in both successors and therefore \( \sigma'_1 \equiv_{\prec_x} \sigma'_2 \).

- Assume that the command at location \( l \) is either “if \( B \) then \( S_1 \) else \( S_2 \) fi” or “while \( B \) do \( S \) od”. Since \( x \) does not appear in \( B \) we know that \( \sigma_1 \models B \Leftrightarrow \sigma_2 \models B \). Also, in all successor states the values of variables do not change. Therefore the successor states of \((l, \sigma_1)\) and \((l, \sigma_2)\) will have the same location, and the same assignments \( \sigma_1 \) and \( \sigma_2 \) for which we know that \( \sigma_1 \equiv_{\prec_x} \sigma_2 \).

We can now prove theorem 4.1 by creating a bisimulation relation [23] on the states of the transition system representing our program. The resulting relation will hold pairs of states which are equivalent with respect to CTL×.

**Proof:** Let \( x \) be a non-visible program variable. We build a relation \( H = H_1 \cup H_2 \subseteq S \times S \) such that

\[
H_1 = \{( (l, \sigma_1), (l, \sigma_2) ) \mid x \text{ is dead at } l \text{ and } \sigma_1 \equiv_{\prec_x} \sigma_2 \}
\]

\[
H_2 = \{( (l, \sigma), (l, \sigma) ) \mid x \text{ is not dead at } l \}
\]

For every \((s_1, s_2) \in H\) we need to prove three things:

1. \( L(s_1) = L(s_2) \)
2. For every \( s_1' \) s.t. \( s_1 \rightarrow s_1' \) there exists a state \( s_2' \) s.t. \( s_2 \rightarrow s_2' \) and \((s_1', s_2') \in H\).
3. For every \( s_2' \) s.t. \( s_2 \rightarrow s_2' \) there exists a state \( s_1' \) s.t. \( s_1 \rightarrow s_1' \) and \((s_1', s_2') \in H\).

For every pair \(( (l, \sigma_1), (l, \sigma_2) ) \in H \) it holds that \( \sigma_1 \equiv_{\prec_x} \sigma_2 \) and, since \( x \) is not visible, \( L(l, \sigma_1) = L(l, \sigma_2) \). It remains to prove the last two conditions. The case when \((s_1, s_2) \in H_2\) is trivial because then \( s_1 = s_2 \). The interesting case then is for \((s_1, s_2) \in H_1\) where \( s_1 = (l, \sigma_1) \) and \( s_2 = (l, \sigma_2) \). Here there are three possibilities:

- \( x \) is neither used nor defined by the command at \( l \). By lemma C.1, for every state \((l', \sigma'_1)\) s.t. \( (l, \sigma_1) \rightarrow (l', \sigma'_1) \) there is a state \((l', \sigma'_2)\) s.t. \( (l, \sigma_2) \rightarrow (l', \sigma'_2) \) and \( \sigma'_1 \equiv_{\prec_x} \sigma'_2 \). By definition, this implies that \((l', \sigma'_1), (l', \sigma'_2) \) \( \in H \). The same holds for the other direction.

- The command at \( l \) is an assignment to \( x \) of an expression \( e \) that does not depend on \( x \) (otherwise \( x \) would not be dead). Since \( \sigma_1(e) = \sigma_2(e) \), if \((l', \sigma'_1)\) is the (only) successor state of \((l, \sigma_1)\) then this is also the (only) successor state of \((l, \sigma_2)\). Obviously, \((l', \sigma'_1), (l', \sigma'_2) \) \( \in H \).

- The command at \( l \) is a non-deterministic assignment to \( x \) of the set of expressions \{\( e_1, \ldots, e_n \)\} such that none of the expressions depend on \( x \). For every expression \( e_i \) chosen, we know that \( \sigma_1(e_i) = \sigma_2(e_i) \), and so the state created by the assignment of \( \sigma_1(e_i) \) into \( x \) is a successor of both \((l, \sigma_1)\) and \((l, \sigma_2)\).

This concludes the proof of theorem 4.1. \( \square \)