State Space Reduction using Dead Variables

Karen Yorav Orna Grumberg
Computer Science Department
The Technion
Haifa 32000, Israel
email: {laster,orna}@cs.technion.ac.il

June 9, 1999

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 gives another approach to handling it.

This work presents a method that uses static analysis of programs to create reduced models of the programs. 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.

We say that a variable is *dead* at a certain point in the code if on all computations from that point on it will be assigned into before its value is read. This means that the value stored in the dead variable can not influence the computation. We use this information in order to reduce the state space of the program by ignoring variable values when the variables are dead.

Further more, we expand the 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 dead(l) for every program location l so that if dead(l) is true than the value of x at the program location l can be discarded. Otherwise, we keep the value of x as usual.

The conditions 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. The smaller transition graph is equivalent to the original transition graph of the program, i.e. a given specification is true in the original transition graph iff it is true in the reduced one. The specifications we consider are formulas of the logic CTL* which is considered a powerful temporal logic, capable of describing important attributes of systems, and especially reactive systems.

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 into one state-transition graph. This solution thus serves to reduce the exponential blow-up that occurs when taking the cross product of the transition graphs 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 process. Our method can be used for verification using an explicit representation of the transition system as well as for verification using a BDD [1] representation. In either case, the verification algorithm itself is not changed, it just receives a (possibly) smaller model. When the transition graph is represented explicitly, as a graph, we are guaranteed to reduce space consumption. Because of the properties of BDDs, when the state-transition graph is represented symbolically, we cannot predict how the reduction will influence the size of the representation.

Our reduction is closely related to partial order reductions [5, 7, 6, 2]. 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 restricts the search of the transition system, but in a different way. During the search we exclude some of the successors of states in which a variable is dead.

We used Murphi [3] 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 a Murphi description of the reduced transition relation created by our method. 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, as expected, smaller than the original.

The paper is organized as follows. Section 2 gives some preliminary definitions which are needed later on. Section 3 presents our reduction method that uses dead variables. Section 4 explains how the dead-variable information can be used to make model checking more efficient. Finally, Section 5 gives the results of using our reduction on an example.

2 Basic Definitions

2.1 Models and Specifications

Kripke structures are widely used for modeling finite-state systems. In this work we use Kripke structures to model the behavior of a finite-state program.

We assume a set AP of *atomic propositions*, which represent the basic state attributes used in specifications. Every atomic proposition is either true or false in a given state.

Definition 2.1: A Kripke structure is a tuple $K = \langle S, I, R, L \rangle$ where S is a (finite) set of states, R a set of edges between states (also called the transition relation), $I \subseteq S$ is a set of initial states, and $L : S \to 2^{AP}$ is a function that associates every state with the set of atomic propositions which are true in that state. We assume that the transition relation is always total, i.e. every state has an out-going edge.

A *path* in K from a state s_0 is an infinite sequence $\pi = s_0, s_1, \ldots$ s.t. for all $i \geq 0$, $s_i \in S$ and $(s_i, s_{i+1}) \in R$. Given a path $\pi = s_0, s_1, \ldots$ we use the notation $\pi^i = s_i, s_{i+1}, \ldots$.

We use CTL* as a specification language.

Definition 2.2: We define *state formulas*, which are formulas that are evaluated over a state, and *path formulas*, which are evaluated over paths. The specification language CTL* is the set of state formulas.

- State Formulas are:
  - Every $p \in AP$.
  - $\varphi_1 \vee \varphi_2$ and $\neg\varphi_1$ for state formulas φ_1, φ_2.
  - $E\psi$ for a path formula ψ.
- Path Formulas are:
  - Every state formula is also a path formula.
  - $\psi_1 \vee \psi_2$ and $\neg\psi_1$ for path formulas ψ_1, ψ_2.
  - $\mathbf{X}\,\psi_1$ and $\psi_1\,\mathbf{U}\,\psi_2$ for path formulas ψ_1, ψ_2.

Let s be a state in some Kripke structure K. The semantics of state formulas is defined by:

- $s \models p$ for $p \in AP$ iff $p \in L(s)$.
- $s \models \varphi_1 \vee \varphi_2$ iff $s \models \varphi_1$ or $s \models \varphi_2$.
- $s \models \neg\varphi_1$ iff $s \not\models \varphi_1$.

- $s \models E\psi$ iff there exists a path π in K that starts from s and $\pi \models \psi$.

Let $\pi = s_0, s_1, \ldots$ be a path in K. The semantics of path formulas is defined by:

- $\pi \models \varphi$ for a state formula φ iff $s_0 \models \varphi$.
- $\pi \models \psi_1 \vee \psi_2$ iff $\pi \models \psi_1$ or $\pi \models \psi_2$.
- $\pi \models \neg\psi_1$ iff $\pi \not\models \psi_1$.
- $\pi \models X\psi$ iff $\pi^1 \models \psi$.
- $\pi \models \psi_1 \mathbf{U} \psi_2$ iff there exists $i > 0$ such that $\pi^i \models \psi_2$ and for every $0 \leq j < i$ it holds that $\pi^j \models \psi_1$.

2.2 Non-deterministic While Programs

The grammar of our programs is extremely simple, but consists of all the constructs necessary so that it can stand for any "real" programming language.

The grammar of NWP programs is:

```
P →     skip |
        x := expr |
        a[expr_1] := expr_2 |
        x := {expr_1, ..., expr_n} |
        a[expr_0] := {expr_1, ..., expr_n} |
        P_1 ; P_2 |
        if B then P_1 else P_2 fi |
        while B do P_1 od
```

where x is a simple variable (boolean or integer), a is a (one dimensional) array 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.

In order to perform model checking on a program we translate it into a Kripke structure.

Let V be the set of program variables, and let Σ be the set of all possible assignments of values to the program variables. Every assignment $\sigma \in Sigma$ is a function that gives each variable $v \in V$ a value $\sigma(v)$ from its domain.

Definition 2.3: Given an NWP program P, its semantics is defined by the Kripke structure $K(P) = \langle S, I, R, L\rangle$ that represents its behavior. Let Loc be the set of program locations (possible positions of the program counter) of P. The set of states is $S = \{(l, \sigma) \mid l \in Loc \wedge \sigma \in \Sigma\}$. The transition relation is defined according to the usual semantics of the commands. We view the evaluation of a boolean expression in an "if" or "while" command as a step in the execution. The set of initial states is the set of states having the location of the beginning of the program. The atomic propositions are assumed to be expressions over V. The labeling function is defined s.t. $L(l, \sigma) = \{p \in AP \mid \sigma \models p\}$.

The resulting structure will be non-deterministic if there is a non-deterministic assignment in the program.

3 Reductions according to Dead Variables

We now present a reduction of the Kripke structures for programs. We create smaller structures for programs than the ones defined in the previous section.

3.1 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 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 3.4: A program variable x is said to be *dead* at a program location l if on every execution path from l, x is assigned a value 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 3.1 bellow.

Definition 3.5: Let Σ 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 $y \neq x$.

We say that x appears in AP if there is an expression in AP that references x.

Theorem 3.1 *Let l be a program location in S, and x a program variable which is dead at l and does not appear in AP. For any two states $(l, \sigma), (l, \sigma')$ s.t. $\sigma \equiv_{-x} \sigma'$ it holds that $(l, \sigma) \equiv_{CTL*} (l, \sigma')$.*

In order to prove Theorem 3.1 we use the following lemma.

Lemma 3.1: Let x be a variable and l a location such that x is not referred to in the command at l. This means that either l is an assignment which does not involve x (on either side), or l is a branching command ("if" or "while") for which the condition does not involve x. 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', σ_1') such that $(l, \sigma_1) \rightarrow (l', \sigma_1')$ there exists a state (l', σ_2') such that $(l, \sigma_2) \rightarrow (l', \sigma_2')$ and $\sigma_1' \equiv_{-x} \sigma_2'$.

Proof: The proof is according to the command at location l.

- Assume that the next command to be executed 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', σ_1') be the successor of (l, σ_1) and let (l', σ_2') be the successor of (l, σ_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 next command to be executed is a non-deterministic assignment "$y := \{e_1, \ldots, e_n\}$". Each successor state (l', σ_1') of (l, σ_1) is a result of assigning one of the expressions e_i to y. Since x does not appear in any of these expressions, there must be an equivalent successor (l', σ_2') of (l, σ_2) which is the result of assigning the same e_i to y. Finally, $\sigma_1(e_i) = \sigma_2(e_i)$ implies that $\sigma_1' \equiv_{-x} \sigma_2'$.

- Assume that the next command to be executed 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, σ_1) and (l, σ_2) will have the same location, and the same assignments σ_1 and σ_2 for which we know that $\sigma_1 \equiv_{-x} \sigma_2$.

□

We can now prove theorem 3.1 by creating a bisimulation relation [4] on the states of the Kripke structure representing our program. The resulting relation will hold pairs of states which are equivalent with respect to CTL*.

Proof: Let x be a program variable that does not appear in AP. We build a relation $H = H_1 \cup H_2 \subseteq S \times S$ such that

$$H_1 = \{((l, \sigma), (l, \sigma')) \mid x \text{ is dead at } l \text{ and } \sigma \equiv_{-x} \sigma'\}$$

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

For every $(s, s') \in H$ we need to prove three things:

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

For every pair $((l, \sigma), (l, \sigma')) \in H$ it holds that $\sigma \equiv_{-x} \sigma'$ and, since x does not appear in AP, $L(l, \sigma) = L(l, \sigma')$. It remains to prove the last two conditions. The case when $(s, s') \in H_2$ is trivial because then $s = s'$ and all the successors of s will also appear in H with themselves. The interesting case then is for $(s, s') \in H_1$ where $s = (l, \sigma)$ and $s' = (l, \sigma')$. Here there are two possibilities:

- x does not appear in the command at l. From lemma 3.1, for every state (l_1, σ_1) s.t. $(l, \sigma) \to (l_1, \sigma_1)$ there is a state (l_1, σ_1') s.t. $(l, \sigma') \to (l_1, \sigma_1')$ and $\sigma_1 \equiv_{-x} \sigma_1'$. By definition, this implies that $((l_1, \sigma_1), (l_1, \sigma_1')) \in H$. 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 $\sigma(e) = \sigma'(e)$, if (l_1, σ_1) is the (only) successor state of (l, σ) then this is also the (only) successor state of (l, σ'). Obviously, $((l_1, \sigma_1), (l_1, \sigma_1)) \in H$.

- 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 $\sigma(e_i) = \sigma'(e_i)$, and so the state created by the assignment of $\sigma(e_i)$ into x is a successor of both (l, σ) and (l, σ').

This concludes the proof of theorem 3.1. $\square$

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

Definition 3.6: We choose a representative value d from the domain of x. Given a program P, the reduced model of P is the Kripke structure $K_R(P)$. This structure is built so that for every state $s = (l, \sigma)$ where x is not dead at l, or x is assigned into at l, the successors of s are the same as in $K(P)$. If x is dead at l and is not assigned into, then for every successor $s' = (l', \sigma')$ of s in K(P) the successor in $K_R(P)$ will be $(l', \sigma'[x \leftarrow d])$.

The reduced structure $K_R(P)$ can be created statically, from the text of P, without building the structure $K(P)$. The reduced structure will have less reachable states, since every equivalence class from H_1 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.

3.2 Partially Dead Variables

We wish to make our reduction more effective (i.e. create even smaller structures) by taking into account more information about the possible use of variables. We notice that in some cases, even though a variable x is not dead at a certain location l, there are possible computations from l on which x will not be used. For example, in figure 1 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, σ) such that $\sigma \not\models (y < 0)$ we can be sure that on every computation that starts from s, x is defined before it is used, i.e. x is dead. 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. Before, we would only have *true* or *false*.

Definition 3.7: Let x be a program variable, l a program location, and σ an assignment to the program variables. We say that x is *dead* at the state $s = \langle l, \sigma \rangle$ if on all

```
l_1:  if (y < 0) then
l_2:        y := x;
      else
l_3:        y := 0;
      fi;
l_4:  x := 0;
```

Figure 1: An example of a partially dead variable

possible runs from s the value of x in s (which is $\sigma(x)$) 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).

As before, if x is dead at s, and x does not appear in the specification, then the value of x at this point is irrelevant. Therefore, if $\sigma \equiv_{-x} \sigma'$ then the two states (l, σ) and (l, σ') must be equivalent with respect to the specification, and all its sub-formulas (since they differ only on the irrelevant value of x). We exploit this fact in order to reduce the transition relation of the program to be verified. Notice that if x appears in the specification (i.e. it is a part of one of the atomic formulas) then these states are not necessarily equivalent.

For the remainder of this paper 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 σ it holds that if $\sigma \models \text{dead}(l)$ then x is dead at (l, σ). The condition we calculate will not be accurate, which means that the implication in the other direction might not be true (i.e. if x is dead at (l, σ) we cannot be sure that $\sigma \models \text{dead}(l)$).

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 dead(l) is not an array.

We need to calculate dead(l) for every program location, and we do it by traversing the *control-flow graph* of P, bottom up. Instead of computing dead(l) directly, we compute for each program location l two conditions: used(l) and def(l). The condition used(l) is created so that if there is a computation from (l, σ) on which x is used (referenced) before it is defined then $\sigma \models \text{used}(l)$. However, there may be states that satisfy the condition, but do not have an outgoing computation on which x is used before it is defined. We compute an over-approximation because calculating the exact set of assignments such that x is used before it is defined cannot be done in a single traversal of the control-flow graph. The condition def(l) is created so that if $\sigma \models \text{def}(l)$ then it is guaranteed that on all computations from (l, σ), x will be defined, and this definition will occur before the first use of x. This time we calculate an under-approximation, so that it is possible that x will not be defined before it is used, and yet $\sigma \not\models \text{def}(l)$.

The conditions def(l) and used(l) are computed together, and then dead(l) is defined as the negation of used(l). In order to compute the conditions we traverse the structure of the program, so that at each step when we calculate the conditions for a sub-program we have already calculated the conditions for its end location.

The first step is to assign conditions to the final program location end: def(end) $= false$, used(end) $= false$. We now describe how to calculate the conditions for a sub-program, given that we have already calculated them for its end location.

- For the sub-program $\underline{l\text{: skip } l'}$:
  def(l) = def(l') and used(l) = used(l').

- For the sub-program $\underline{l\text{: } x := exp\ l'}$:
  If the expression exp does not use x then def(l) $= true$ and used(l) $= false$. If exp does use x then def(l) $= false$ and used(l) $= true$.

- For the sub-program $\underline{l\text{: } y := exp\ l'}$ ($y \neq x$):
  If the expression exp does not use x then we change our conditions according to the assignment: def(l) = def(l')$[y \leftarrow exp]$ and used(l) = used(l')$[y \leftarrow exp]$. If exp does use x then def(l) $= false$ and used(l) $= true$.

- For the sub-program $\underline{l\text{: } y := \{exp_1, \ldots, exp_n\}\ l'}$:
  We add up the influences of all the possible assignments. If x is used by one of the expressions $exp_1, \ldots, exp_n$ or if $y = x$ then the assignment either uses or defines x, and the conditions are defined accordingly (as was done in the regular assignment cases above). Otherwise, def(l) $= \bigwedge_{j=1,\ldots,n}$ def(l')$[y \leftarrow exp_j]$ and used(l) $= \bigvee_{j=1,\ldots,n}$ used(l')$[y \leftarrow exp_j]$.

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

- For the sub-program $\underline{\text{l: if } B \text{ then } l_1\text{: } S_1 \text{ else } l_2\text{: } S_2 \text{ fi } l_e}$:
  We calculate by a recursive call the conditions def(l_i) and used(l_i) for $i = 1, 2$, using def(l_e) and used(l_e) as input for both calculations. If the condition B does not use x we can set def(l) $= (B \wedge$ def$(l_1)) \vee (\neg B \wedge$ def$(l_2))$ and used(l) $=$ $(B \wedge$ used$(l_1)) \vee (\neg B \wedge$ used$(l_2))$. If, on the other hand, the condition B uses the variable x then def(l) $= false$ and used(l) $= true$.

- For the sub-program $\underline{l\text{: while } B \text{ do } l_1\text{: } S_1\ l_1' \text{ od: } l_e}$:
  Similarly to the "if" case, if B uses x then def(l) $= false$ and used(l) $= true$.

  Otherwise, we use a recursive call to calculate def(l_1) and used(l_1). The input to this call is the "safest" approximation we can give, since we do not have any information on what happens at the end of the body. If x does not appear in S_1 at all, which can be checked while parsing the program, and if used(l_e) $= false$

then we assume used(l'_1) $= false$. Otherwise, we assume used(l'_1) $= true$. In both cases we assume def(l'_1) $= false$.

When the recursive call for S_1 is done we define:

$$\text{def}(l) = (B \wedge \text{def}(l_1)) \vee (\neg B \wedge \text{def}(l_e))$$

$$\text{used}(l) = \begin{cases} false & \text{if used}(l_1) = \text{used}(l_e) = false \\ (B \wedge \neg\text{def}(l_1)) \vee (\neg B \wedge \text{used}(l_e)) & \text{otherwise} \end{cases}$$

There are several optimizations that can be made to the above definition, in order to get better approximations of *used* and *defined*. 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 *used* and *defined*. Therefore an optimization of the calculation of *used* and *defined* will only change the way we calculate them for a loop.

One optimization, which we used in our examples, is to traverse each loop twice so that we can identify situations in which x is never used at the top of the body. In order to have used(l_1) $= false$ according to the above algorithm we would have to have def(l_1) $= true$ since we process S_1 under the assumption used(l'_1) $= true$. Instead, we propose to first compute *used* and *defined* on S_1 under the assumptions used(l'_1) $=$ def(l'_1) $= false$. If under these conditions we find that used(l_1) $= false$ then we can conclude that x can never be used before it is defined inside the loop. We can therefore set used(l) $= false$. If, on the other hand, the calculation result is that used(l_1) $\neq false$ then we have calculated our conditions for S_1 under false assumptions, and therefor cannot use our results. We then do another round on S_1, this time with the assumptions mentioned above - used(l'_1) $= true$ and def(l'_1) $= false$.

Notice that the conditions used(l) and def(l) can never be dependent on x. When one of these conditions 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 used(l) $= true$ and def(l) $= false$.

3.3 Parallel Programs

A parallel program $P = [P_1, \ldots, P_n]$ is a parallel composition of sequential programs. Each process P_i is a non-deterministic while program, as defined in section 2. Every process has its own local variables, and is not allowed to assign to or read from local variables of other processes. We add to our language *send* and *receive* commands so that the command $send(P_i, e)$ (inside the body of P_j) sends the value of the expression e to the process P_i, and the command $receive(P_j, x)$ (inside the body of P_i) receives from the process P_j a value to be assigned into x. The variables that appear in e must be local variables of P_j, and the variable x must be a local variable of P_i.

We augment the computation of used(l) and def(l) with instructions for handling communication commands:

- For the sub-program $\underline{l\text{: send}(P_i,exp)\ l'}$:
  If exp uses x then def(l) $= false$ and used(l) $= true$. Otherwise, def(l) = def(l') and used(l) = used(l').

- For the sub-program $\underline{l\text{: receive}(P_i,x)\ l'}$:
  def(l) $= true$ and used(l) $= false$.

- For the sub-program $\underline{l\text{: receive}(P_i,y)\ l'}$ $(y \neq x)$:
  def(l) $= \forall y$.def(l') and used(l) $= \exists y$.used(l').

This allows us to compute, for each process separately, the condition dead(l) for every location of the process. We create a reduced model for each process, and only then create the cross product of the models to get a model for the full program.

Theorem 3.2 *Let $P = [P_1|| \ldots ||P_n]$ be a parallel program. Then the parallel composition of the reduced structures for the processes is bisimilar to the parallel composition of the original structures. Formally, if $K(P) = K(P_1)|| \ldots ||K(P_n)$ and $K_R(P) = K_R(P_1)|| \ldots ||K_R(P_n)$, then $K(P) \equiv_{CTL*} K_R(P)$.*

Proof: Let H_i be the bisimulation relation that was defined earlier between $K(P_i)$ and $K_R(P_i)$. Every state s in $K(P)$ is a tuple $(s_1, \ldots, s_n)$ such that s_i is a state of $K(P_i)$, and similarly every state t in $K_R(P)$ 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. □

4 Application to Explicit-State Model Checking

Explicit-State Model Checking techniques are techniques that keep the Kripke structure of the system to be verified as a graph. In general, the specification is checked by traversing this graph state by state. The state explosion problem manifests itself in the size of this graph, which is typically too large to be kept in the memory of the computer. There exist some methods by which this graph need not be kept in whole, and the exploration proceeds via a "next-state function". This function is given a state, and produces the set of sons of this state. However, in order to perform CTL model checking one needs to keep track of the path (or paths) that was traversed until the current state was encountered. This path may in itself be extremely long, so that the algorithm may not be able to keep it in the computer memory, and the model checking fails. It is therefore useful to be able to reduce the number of states in the Kripke structure to be checked.

Given a program P and a choice of variable x (that does not appear in the specification), we compute dead(l) for all the locations in P. We now alter the next-state function so that whenever it gets a state (l, σ) s.t. $\sigma \models$ dead(l) and the statement at l does not assign into x, it reduces the set of next-states to only those in which $x = 0$, or any other value (if x is boolean - we choose either $true$ or $false$). 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 Kripke structure. The previous section showed that the pruned structure is CTL* equivalent to the original structure, and therefore the result of the model checking algorithm is not altered.

5 Experimental Results

We chose as an example an implementation of a distributed linked list for sorting. Each process receives a number from its left, compares it to its own number, and sends to the right the larger of the two (keeping the smaller). In this way, when there are no more messages sent, the numbers are sorted. We examined two versions, one with 3 nodes (and numbers in the range 0-2) and the other with 4 nodes (and numbers in the range 0-3).

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 calculated the condition dead(l) for the variable tmp and every location l. The semantics of commands was changed according to this information. For example, whenever tmp changes from "live" to "dead", it is assigned a designated value (zero).

Figure 5 shows the resulting number of states in the original model and in the reduced model for our example. The table also gives the number of edges in the model, and the time it took Murphi to traverse the graph. The numbers in parenthesis give the percentage of reduction in size of the reduced model w.r.t the full model. They are calculated by the formula: 100 - (reduced \ full)*100). The examples were run using a 400M hash table.

	No. of States	No. of Edges	Time (sec.)
Full Model 3 nodes	35796	100791	22
Reduced Model 3 nodes	(5%) 33996	(4%) 96597	(0%) 22
Full Model 4 nodes	1203536	4147621	659
Reduced Model 4 nodes	(12%) 1054448	(11%) 3679149	(22%) 512

Figure 2: Results of reductions

References

[1] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[2] Patrice Godefroid. *Partial order methods for the verification of concurrent systems.* PhD thesis.

[3] Murphi description languange and verifier. The URL for the home page of murphi is: http://sprout.stanford.edu/dill/murphi.html.

[4] D. Park. Concurrency and automata on infinite sequences. In *5th GI-Conference on Theoretical Computer Science*, pages 167–183. Springer-Verlag, 1981. LNCS 104.

[5] D. Peled. Ten years of partial order reduction. In A. J. Hu and M. Y. Vardi, editors, *Proc. of the 10th International Conference on Computer Aided Verification, LNCS 1427*, 1998.

[6] A. Valmari. A stubborn attack on the state explosion problem. In R. P. Kurshan and E. M. Clarke, editors, *Proceedings of the 1990 Workshop on Computer-Aided Verification*, June 1990.

[7] P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In *Proceedings of CUNCUR'93*, volume 715 of *LNCS*, pages 233–246. Springer Verlag, 1993.