Automatically Closing Open Reactive Programs

C. Colby, P. Godefroid and L. J. Jagadeesan

June 1998

Proceedings of 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation

Automatically Closing Open Reactive Programs

Christopher Colby
Loyola University Chicago
Dept. of Math. and Comp. Sciences
6525 N. Sheridan Rd.
Chicago, IL 60626
colby@math.luc.edu

Patrice Godefroid
Bell Laboratories
Lucent Technologies
1000 E. Warrenville Road
Naperville, IL 60566
god@bell-labs.com

Lalita Jategaonkar Jagadeesan
Bell Laboratories
Lucent Technologies
1000 E. Warrenville Road
Naperville, IL 60566
lalita@bell-labs.com

Abstract

We study in this paper the problem of analyzing implementations of open systems — systems in which only some of the components are present. We present an algorithm for automatically closing an open concurrent reactive system with its most general environment, i.e., the environment that can provide any input at any time to the system. The result is a nondeterministic closed (i.e., self-executable) system which can exhibit all the possible reactive behaviors of the original open system. These behaviors can then be analyzed using VeriSoft, an existing tool for systematically exploring the state spaces of closed systems composed of multiple (possibly nondeterministic) processes executing arbitrary code. We have implemented the techniques introduced in this paper in a prototype tool for automatically closing open programs written in the C programming language. We discuss preliminary experimental results obtained with a large telephone-switching software application developed at Lucent Technologies.

1 Introduction

Systematic state-space exploration, as such or elaborated into temporal-logic model-checking (e.g., [CES86, QS81]), is attracting growing attention for checking the correctness of concurrent reactive systems. State-space exploration tools for software systems have traditionally been restricted to the exploration of the state space of an abstract description of the system, specified in a modeling language (e.g., [Hol91, CPS93, McM93]). VeriSoft [God97] is a recent tool which extends the scope of systematic state-space exploration to concurrent systems in which processes execute arbitrary code written in general-purpose programming languages such as C or C++. VeriSoft explores the state space of a concurrent system by actually executing the code for the components of the system and by controlling their synchronizations. It thus eliminates the need for modeling the system, an often time-consuming and error-prone task, and combines aspects of debugging and replay tools for concurrent systems with the sort of state-space exploration associated with model checking.

Systematic state-space exploration requires the system being analyzed to be closed, i.e., self-executable. This implies that, given an open system, an executable representation of the environment in which the system operates need be available for closing the system. Providing *manually* a simple but faithful representation of the environment of an open system is often a difficult task. In order to analyze the implementation of a large open system with VeriSoft, this task can become tedious and impractical because the interface between the implementation of a system and its environment can be complex. We address in this paper the problem of *automatically* closing the implementation of an open system.

Given an open system, a model of its environment — whether generated manually or automatically — might be either too restrictive, in which case it may cause the verification to miss some possible erroneous behaviors of the system, or too general, in which case it will increase the size of the state space and may result in unrealistic erroneous behaviors. This paper describes an algorithm for closing a system with its most general environment. In this way, it is certain that the verification will not miss any erroneous behaviors due to an insufficiently general environment. In practice, the intended use of this approach is that a developer provides manually an implementation for a *partial* model of the environment, in order to capture more precisely certain areas of interest, and then applies our algorithm to close the remainder of the system.

At first blush, such an algorithm might seem easy to achieve: Given an open system $\mathcal{S}$, add a new component $E_{\mathcal{S}}$ to $\mathcal{S}$ whose behavior includes all possible sequences of inputs and outputs of $\mathcal{S}$. However, this naive approach generates a closed system whose state space is typically so large that it renders any analysis intractable: for instance, $E_{\mathcal{S}}$ is infinitely branching whenever the set of inputs is infinite. Instead, our algorithm uses a *static analysis* of the open system to *eliminate its interface*. In fact, our transformation preserves, or may even reduce, the static degree of branching of the original code.

Our work is inspired by challenges encountered when applying VeriSoft to analyze software applications in Lucent Technologies' 5ESS telephone switching system. Many applications within the 5ESS software are composed of concurrent reactive components of considerable size and complexity, and cannot be suitably analyzed using traditional testing tools. On the other hand, it is non-trivial to close such multi-process applications in order to analyze them using automatic state-space exploration techniques. For example, manually closing a 5ESS application can be tantamount to simulating millions of lines of code implementing the rest of the 5ESS software.

We have implemented the techniques introduced in this paper in a prototype tool for automatically closing open programs written in the C programming language. We have applied our tool to a complex 5ESS software application, and are in the process of analyzing the resulting closed system using VeriSoft.

This paper is organized as follows. In the next section, we give some background about the verification of concurrent reactive systems, and describe the underlying framework adopted in this work. Then we discuss in Section 3 basic issues involved in closing open concurrent reactive systems. Next, we present our algorithm for closing open programs, and illustrate it with two examples. The correctness of the algorithm is established in Section 5, where its precision is also discussed. We then remark on our practical experience in applying this work to telecommunication software. Finally, we conclude with a comparison between program closing and related work, and with suggestions on possible improvements.

2 Background

We recall in this section the framework introduced in [God97]. We consider a closed concurrent system $\mathcal{S}$ composed of a finite set $\mathcal{P}$ of *processes* and a finite set $\mathcal{O}$ of *communication objects*. Each process $P \in \mathcal{P}$ executes a sequence of *operations* that is described in a sequential program written in a full-fledged programming language such as C. Such sequential programs are *deterministic*: every execution of the program on the same data performs the same sequence of operations. We assume that processes only communicate with each other by performing operations on communication objects. A communication object $O \in \mathcal{O}$ is defined by a pair (V, OP), where V is the set of all possible values for the object (its domain), and OP is the set of *operations* that can be performed on the object. Examples of communication objects are shared variables, semaphores, and FIFO buffers. At any time, at most one operation can be performed on a given communication object (operations on a same communication object are mutually exclusive). For the purpose of this work, we also assume that the enabledness of any operation on any communication object depends exclusively on the sequence of operations that has been performed on the object in the history of the system, and not on values that are possibly stored or passed through the object, or passed as argument to the operation. Operations on communication objects are called *visible operations*, while other operations are by default called *invisible*. The execution of an operation is said to be *blocking* if it cannot be completed. We assume that only executions of visible operations may be blocking.

At any time, the concurrent system is said to be in a *state*. The system is said to be in a *global state* when the next operation to be executed by every process in the system is a visible operation. We assume that every process in the system always eventually attempts to execute a visible operation.[1] This implies that initially, after the creation of all the processes of the system, the system may reach a first and unique global state s_0, called the *initial global state* of the system. We define a *process transition*, or *transition* for short, as one visible operation followed by a *finite* sequence of invisible operations performed by a single process and ending just before a visible operation. Let $\mathcal{T}$ denote the set of all transitions of the system.

A transition is said to be *disabled* in a global state s when the execution of its visible operation is blocking in s. Otherwise, the transition is said to be *enabled* in s. A transition t that is enabled in a global state s can be *executed* from s. Because the number of invisible operations in a transition is finite, the execution of an enabled transition always terminates. When the execution of t from s is completed, the system reaches a global state s', called the *successor* of s by t.[2] We write $s \xrightarrow{t} s'$ to mean that the execution of the transition t leads from the global state s to the global state s', while $s \overset{w}{\Rightarrow} s'$ means that the execution of the finite sequence w of transitions leads from s to s'. If $s \overset{w}{\Rightarrow} s'$, s' is said to be *reachable* from s.

We now define a formal semantics for the concurrent systems that satisfy our assumptions. A concurrent system as defined here is a closed system: from its initial global state, it can evolve and change its state by executing enabled transitions. Therefore, a natural way to describe the possible *behaviors* of such a system is to consider its set of reachable global states and the transitions that are possible between these.

Formally, the joint *global* behavior of all processes in a concurrent system can be represented by a transition system $A_{\mathcal{S}} = (S, \Delta, s_0)$ such that S is the set of global states of the system, $\Delta \subseteq S \times S$ is the *transition relation* defined such that $(s, s') \in \Delta$ iff $\exists t \in \mathcal{T} : s \xrightarrow{t} s'$, and s_0 is the initial global state of the system. An element of Δ corresponds to the execution of a single transition $t \in \mathcal{T}$ of the system. The elements of Δ will be referred to as *global transitions*. It is natural to restrict $A_{\mathcal{S}}$ to its global states and transitions that are reachable from s_0, because the other global states and transitions play no role in the behavior of the system. In what follows, a "state in $A_{\mathcal{S}}$" denotes a global state that is reachable from s_0. $A_{\mathcal{S}}$ is called the *global state space* of the system.

Because we consider here closed concurrent systems, the environment of one process is formed by the other processes in the system. This implies that, in the case of a single "open" reactive system, the environment in which this system operates has to be represented, possibly using other processes. In practice, a complete representation of such an environment may not be available, or may be quite complex. It is then convenient to use a *model*, i.e., a simplified representation, of the environment to simulate its external behavior. For this purpose, we introduce a special operation "VS_toss" to express a valuable feature of modeling languages: *nondeterminism*. This operation takes as argument a positive integer n, and returns an integer in $[0, n]$. The operation is nondeterministic: the execution of a transition starting with VS_toss(n) may yield up to $n+1$ different successor states, corresponding to different values returned by VS_toss. In contrast with what was defined in [God97],

[1] VeriSoft (see forthcoming description) reports a "divergence" when a process does not attempt to execute any visible operation for more than a given (user-specified) amount of time.

[2] Operations on objects (and hence transitions) are deterministic: the execution of a transition t in a state s leads to a *unique* successor state.

we consider VS_toss as an *invisible* operation in this paper, in order to simplify the following presentation and terminology.

In [God97], it is shown that *deadlocks* and *assertion violations* can be detected by exploring only the global states of a concurrent system as defined above. Deadlocks are states where the execution of the next operation of every process in the system is blocking. Assertions can be specified by the user with the special operation "VS_assert". This operation can be inserted in the code of any process, and is considered visible. It takes as its argument a boolean expression that can test and compare the value of variables local to the process. When "VS_assert(expression)" is executed, the expression is evaluated. If the expression evaluates to false, the assertion is said to be *violated*.

VeriSoft [God97] is a tool for systematically exploring the state space of a concurrent system as defined above. In a nutshell, every process of the concurrent system to be analyzed is mapped to a UNIX process. The execution of the system processes is controlled by an external process, called the *scheduler*. The scheduler observes the visible and VS_toss operations performed by processes inside the system, and can suspend their execution. By resuming the execution of (the next visible operation of) one selected system process in a global state, the scheduler can explore one transition between two global states in the state space of the concurrent system. By reinitializing the system, the scheduler can explore alternative paths in the state space. Because states of processes can be complex (due to pointers, dynamic memory allocation, large data structures of various shapes, recursion, etc.), VeriSoft does not attempt to compute any representation for the reachable states of the system being analyzed, and hence performs a systematic state-space exploration without storing any intermediate states in memory. It is shown in [God97] that the key to make this approach tractable is to use a new search algorithm built upon existing state-space pruning techniques known as partial-order methods [God96]. For finite acyclic state spaces, this search algorithm is guaranteed to terminate and can be used for detecting deadlocks and assertion violations without incurring the risk of any incompleteness in the verification results. In practice, VeriSoft can be used for systematically and efficiently testing the correctness of any concurrent system, whether or not its state space is acyclic. Indeed, it can always guarantee, from a given initial state, complete coverage of the state space up to some depth.

3 Closing Open Systems

Systematic state-space exploration requires the system being analyzed to be closed. Given an open system, an executable representation of its environment need be provided for closing the system. In this section, we define more precisely what "closing an open system" means, and discuss several approaches to the problem.

Let us now consider an *open* concurrent system $\mathcal{S}$: $\mathcal{S}$ can interact with its environment via an interface composed of a set $I_{\mathcal{S}}$ of inputs and a set $O_{\mathcal{S}}$ of outputs. Let V_i denote the set of possible input values that can be provided by the environment to the system $\mathcal{S}$ via input i in $I_{\mathcal{S}}$, and let V_o denote the set of possible output values that can be produced by $\mathcal{S}$ to its environment via output o in $O_{\mathcal{S}}$. Closing such an open system means combining the system with an executable representation for its environment. The result of the combination is a self-contained executable system.

Creating an executable representation for the environment of an open system can be a tedious task, because (1) the interface between the system and its environment may be complex and (2) many possible data values may be provided by the environment. In order to facilitate this task, it would be useful to have an algorithm for generating automatically an executable representation of *the most general environment* in which an open system can operate. Precisely, we define the most general environment $E_{\mathcal{S}}$ of a system $\mathcal{S}$ as the environment that nondeterministically provides any value v_i in V_i whenever the system $\mathcal{S}$ takes an input i in $I_{\mathcal{S}}$, and that can take any output o in $O_{\mathcal{S}}$ produced by the system. By construction, the combination of the system $\mathcal{S}$ with its most general environment $E_{\mathcal{S}}$ makes it possible to observe all the possible visible behaviors that $\mathcal{S}$ can exhibit. Note that, by definition, there are no dependencies between the input and output values provided and accepted by $E_{\mathcal{S}}$, respectively.

Generating automatically $E_{\mathcal{S}}$ from $\mathcal{S}$ would address problem (1) above, but not problem (2): the set of input values provided by $E_{\mathcal{S}}$ can be very large, even infinite, which would yield an intractable state-space search. A better approach would be to partition the sets of possible input values provided by $E_{\mathcal{S}}$ into equivalence classes such that values in the same equivalence class would imply exactly the same behavior of $\mathcal{S}$. Of course, partitioning each data domain V_i, $i \in I_{\mathcal{S}}$, into such equivalence classes is a hard problem, and undecidable in general. Computing simple conservative approximations for these equivalence classes is possible but would require a sophisticated, and hence more expensive, static analysis of the code describing the system. We will come back to this option in Section 7.

In this work, we investigate an alternative approach: Instead of generating an executable representation of the most general environment $E_{\mathcal{S}}$ of an open system $\mathcal{S}$, we propose to *eliminate its interface* altogether. Specifically, we present in the next section an algorithm that transforms an open system $\mathcal{S}$ into a closed (nondeterministic) system $\mathcal{S}'$ such that all data values in $\mathcal{S} \times E_{\mathcal{S}}$ that may depend on the behavior of $E_{\mathcal{S}}$ are eliminated in $\mathcal{S}'$, and all control-flow choices in $\mathcal{S} \times E_{\mathcal{S}}$ that may depend on these data values are replaced by non-deterministic choices in $\mathcal{S}'$. The reactive behavior of $\mathcal{S} \times E_{\mathcal{S}}$ and $\mathcal{S}'$, as well as their effect on data values that do not depend on $E_{\mathcal{S}}$, are closely related:

- Every execution of $\mathcal{S} \times E_{\mathcal{S}}$ corresponds to an execution of $\mathcal{S}'$ that exhibits the same sequence of visible operations and that preserves all data values that do not depend on $E_{\mathcal{S}}$.
- All deadlocks and all assertion violations in $A_{\mathcal{S} \times E_{\mathcal{S}}}$ that evaluate only expressions whose value does not depend on $E_{\mathcal{S}}$ are preserved in $A_{\mathcal{S}'}$.

Our algorithm performs a *static analysis* of the source code executed by the processes P in $\mathcal{P}$ of the open system $\mathcal{S}$. Specifically, we assume that each process P in $\mathcal{P}$ executes a sequence of operations, that is described in a *sequential program* written in a full-fledged programming language such as C. We also assume that such a program can be decomposed into a finite collection of procedures p_j which may call each other, and includes a unique top-level procedure.

Because open systems are composed of processes, and because programs describing processes are composed of procedures, we map the notions of inputs and outputs from the system level to the procedure level as follows. Let I_j and O_j denote the input and output sets of procedure p_j, respectively. When a procedure p_j produces an output o in O_j that is taken as input i in I_k by another procedure p_k, we will write $o = i$. By construction, we have:

$$I_{\mathcal{S}} = (\bigcup_j I_j) \setminus (\bigcup_j O_j) \text{ and } O_{\mathcal{S}} \subseteq \bigcup_j O_j.$$

Examples of input values of a procedure p_j include values passed as argument to p_j when it is being called, and pointers to values used in p_j but set outside of p_j.

In the next section, we present an algorithm for transforming each procedure p_j individually. Then, we prove that these "local" transformations preserve together the visible behavior of each of the processes P in $\mathcal{P}$ and of the system $\mathcal{S}$ itself.

4 The Algorithm

We assume we are working with an imperative programming language that meets the following general description. A program consists of a sequence of *statements* of the following four types: assignment statements which assign *values* to memory locations called *variables*, conditional statements (if–then–else, switch–case, while, for), procedure call statements, and termination statements (return, exit). The programming language also provides basic atomic data structures (e.g., integer, real, boolean), and constructor and selector operations (e.g., records, arrays, pointers) for creating and manipulating data structures built up from the basic data structures. Visible operations are procedure calls of a specific kind.

Throughout this paper, we use the term *variable* to refer to a memory location in which a value may be stored. Examples of variables are identifiers (program variables), pointers, array elements, mutable record fields, and so forth. A variable is thus a semantic object rather than a syntactic one. Our motivation for this is several-fold: increased generality, independence of the framework from the choice of source language, and increased abstraction in the correctness specification of our algorithm.

Every procedure p_j described in a program satisfying the above assumptions can be represented by a *control-flow graph* $G_j = (N_j, A_j)$, where

- the set of nodes N_j is the set of statements that appear in the program describing p_j;
- the set of arcs $A_j \subseteq N_j \times N_j$ is such that $(n, n') \in A_j$ iff the program statement corresponding to n' may be the next one to be executed by p_j after the execution of the program statement corresponding to n.

Each arc (n, n') in A_j is labeled with a boolean expression on variables occurring in the statement that specifies when the program statement corresponding to n' is executed after the program statement corresponding to n. For every node n in N_j, the boolean expressions that label arcs from n are mutually exclusive, and their disjunction is a tautology. Note that we assume that the control flow describing the sequencing of operations performed by a process is completely specified by the procedures p_j; interruptions and other preemptive schemes that may violate this assumption are not considered here for the sake of simplicity.

Let n be a node of the control-flow graph of some procedure p_j implementing an open system $\mathcal{S}$. We say a variable v is *used* in node n if the value of v may be required during some execution of the statement corresponding to n (i.e., the value in the memory location corresponding to v may be read). Similarly, we say a variable v is *defined* in node n if the value of v may be modified during the execution of the statement corresponding to n (i.e., a value may be written in the memory location corresponding to v). Let $V(n)$ denote the set of variables used in node n. For simplicity, we assume that every execution of an assignment statement defines the value of exactly one variable. Conditional and termination statements are assumed not to define any variables.

Procedure calls are modeled as follows. We assume that each argument of a procedure call is a variable. When the procedure call is executed, a new fresh variable is created for each argument and is initialized (defined) with the value of the corresponding variable passed as argument. Fresh variables are assumed not to escape their scope: they cannot be used by the calling procedure. The execution of the called procedure can then start with the execution of its start node. By definition, we assume that start nodes do not use nor define any variables. After the execution of a termination statement in the called procedure, the execution of the calling procedure is resumed at the (unique) node following the corresponding procedure call. We assume that termination statements in the top-level procedure of any process is always blocking. (Therefore, the number of processes is always constant.)

In addition to the above somewhat standard assumptions, we assume that, for each input i in I_j, it is possible to determine whether i is also in $I_{\mathcal{S}}$. This means that it is possible to determine statically which input values of a procedure p_j may be provided by the environment $E_{\mathcal{S}}$, including indirectly via other procedures. For simplicity, we assume the environment $E_{\mathcal{S}}$ is not allowed to define variables that have been previously defined by the system $\mathcal{S}$.

For every procedure p_j, we also compute a *define-use graph* $\overline{G_j} = (N_j, \overline{A_j})$, where

- the set of nodes N_j is the set of statements that appear in the program describing p_j;
- the set of arcs $\overline{A_j} \subseteq N_j \times N_j$ is such that (n, n') is in $\overline{A_j}$ implies that the program statement corresponding to n' *uses* the value of a variable v *defined* by the program statement corresponding to n; the arc (n, n') is then labeled with v. Furthermore, if a node n defines a variable v and a node n' uses variable v, and if there is a control-flow path from n to n' in G_j along which v is not defined, then there is an arc (n, n') in $\overline{A_j}$ labeled with v.

Techniques for computing define-use dependencies are discussed, e.g., in [ASU86, FOW87, MR90], and require a may-alias analysis (e.g., [CWZ90, Lan91, Deu94, Ruf95]).

We now turn to the presentation of our algorithm for closing an open procedure p_j. This algorithm is presented in Figure 1. It takes as input both the control-flow graph G_j and the define-use graph $\overline{G_j}$ of the procedure p_j. The algorithm generates a new control-flow graph G'_j by transforming G_j using information extracted from $\overline{G_j}$. From G'_j,

1. *Input:* **the control-flow graph G_j and define-use graph $\overline{G_j}$ of procedure p_j.**
2. **Analysis of $\overline{G_j}$ to compute $V_I(n)$ for each node n:**
   - **Let N_{E_S} denote the set of nodes in N_j that uses the value of a variable defined by the environment E_S.**
   - **Compute the set N_I of nodes in N_j that are reachable from a node in N_{E_S} by a (possibly empty) sequence of define-use arcs in $\overline{G_j}$.**
   - **For each node n in N_I, let $V_I(n)$ denote the set of variables used in n that are defined by E_S or that are labeling an arc leading to n from $n' \in N_I$ in $\overline{G_j}$.**
   - **For each node n not in N_I, we have $V_I(n) = 0$.**
3. **Mark the nodes of G_j according to the following rules:**
   - **mark the start node;**
   - **mark each node corresponding to a termination statement;**
   - **mark each node corresponding to a call to a procedure of the system;**
   - **mark each node n corresponding to an assignment or conditional statement such that n is *not* in N_I.**
4. **Generate the control-flow graph $G'_j = (N'_j, A'_j)$ as follows.
   For each node n of G_j marked in Step 3, do the following:**
   - **add n to N'_j;**
   - **for each arc $a = (n, n') \in A_j$, let $succ(a)$ denote the set of marked nodes of G_j that are reachable from n by a sequence aw of control-flow arcs in G_j passing through unmarked nodes exclusively and starting with arc a;**
     - **if $|succ(a)| = 0$, do nothing;**
     - **if $|succ(a)| = 1$, add an arc in A'_j from n to the node in $succ(a)$ and labeled with the boolean expression labeling arc a;**
     - **if $|succ(a)| > 1$, create a new node n'' corresponding to a conditional statement testing the value of "VS_toss($|succ(a)| - 1$)"; add an arc in A'_j from n to n'' and labeled with the boolean expression labeling arc a; then, for every node $n_k \in succ(a)$, $0 \leq k \leq (|succ(a)| - 1)$, add an arc in A'_j from n'' to n_k and labeled with a boolean expression that is satisfied iff the value returned by the call to VS_toss performed in n'' returns k.**
5. **Perform the following final modifications to G'_j:**
   - **remove the parameters of p_j that are defined by E_S;**
   - **for each node n in G'_j corresponding to a procedure call to procedure p_l, remove each argument of the procedure call whose corresponding parameter has been removed by Point 1 of Step 5 when transforming G_l.**
6. *Output:* **the control-flow graph G'_j of procedure p'_j.**

Figure 1: Algorithm transforming G_j into G'_j

it is then easy to construct a new procedure p'_j that has G'_j for control-flow graph.

Let us detail the different steps performed by the algorithm. Step 2 determines which program statements of p_j may use (possibly indirectly via other variables) a value defined by the environment E_S. This information is computed from the define-use graph of p_j. Step 3 of the algorithm selects the program statements of p_j that will be preserved in p'_j. These include all the procedure calls (which, by definition, include all the visible operations) and termination statements, as well as the assignment and conditional statements that do not use any value provided by E_S. Then, Step 4 constructs from the control-flow graph G_j a new control-flow graph G'_j which simulates, using the nondeterministic VS_toss operation, all the possible effects of the values provided by E_S on the control-flow of p_j. Step 5 completes the transformation by removing references to parameters of procedures that may be used to transmit values of variables defined by E_S. Note that "variables defined by E_S" from the point of view of a procedure include variables v defined in other procedure calls during the executions of nodes n corresponding to assignment statements such that $V_I(n) \neq \emptyset$, or of nodes n corresponding to procedure calls such that $v \in V_I(n)$. Therefore, the existence of a single node n corresponding to a procedure call to p_j such that $v \in V_I(n)$ is sufficient to make Point 1 of Step 5 remove the parameter of p_j corresponding to v.

The overall time complexity of the above algorithm is essentially linear in the size of G_j and $\overline{G_j}$ since the transformation can be performed by a single traversal of both graphs. Note that Step 4 of the algorithm eliminates cyclic paths that traverse exclusively unmarked nodes. Divergences due to such paths are therefore not preserved in G'_j.

Figures 2 and 3 illustrate the result of applying our algorithm on two different open procedures, p and q. The graphs on the left are the control-flow graphs G_p and G_q of the procedures. In each case, the procedure takes as input a value provided by the environment E_S and stored in a variable named x. The graphs on the right are the closed control-flow graphs G'_p and G'_q that the algorithm generates from G_p and G_q, respectively. Note that G'_p and G'_q are equivalent; although p and q are functionally distinct, the algorithm transforms each of them to the same closed program. In the case of procedure p, the resulting closed program is a strict upper approximation of p combined with its most general environment E_S. For no values of x can G_p send a mixture of "even" and "odd" values, but for certain combinations of VS_toss results, G'_p can. In the case of procedure q, however, the resulting closed program is equivalent to q combined with its most general environment E_S. In this case, q sends the ten least-significant bits of x, and so the set of executions induced by the set of all input values x is equivalent to the set of executions induced by the set of all VS_toss results.

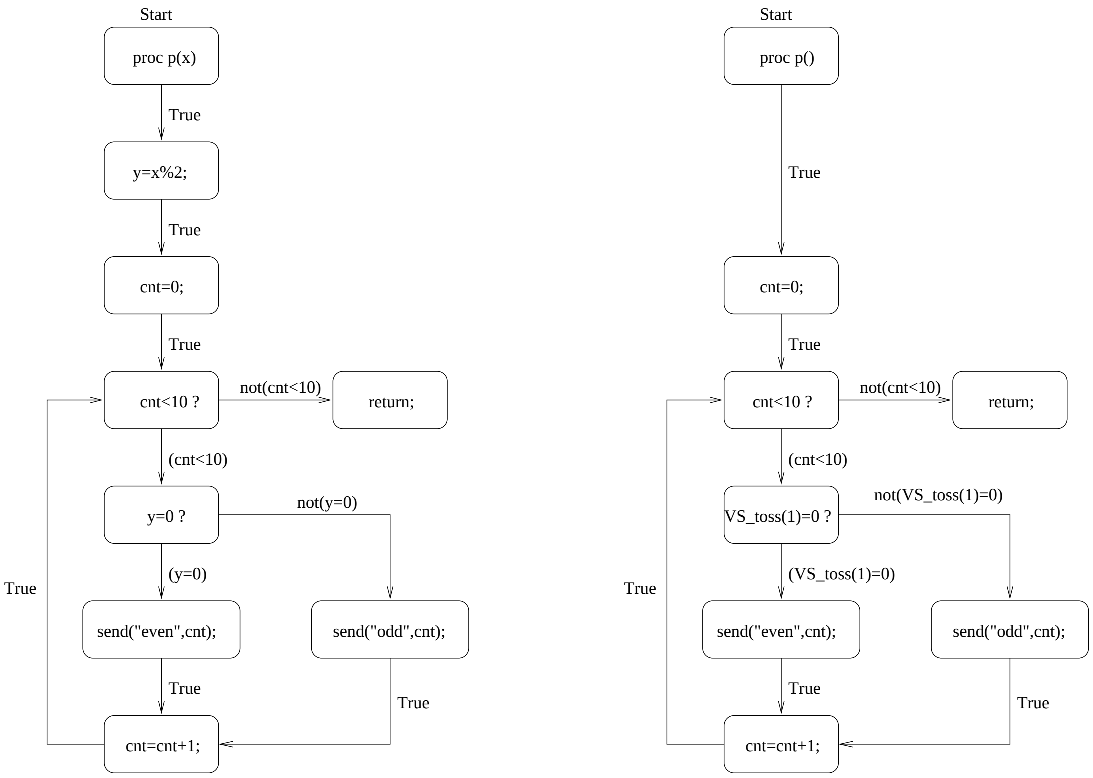

Figure 2: A simple example of transformation: original G_p (left) and transformed G'_p (right)

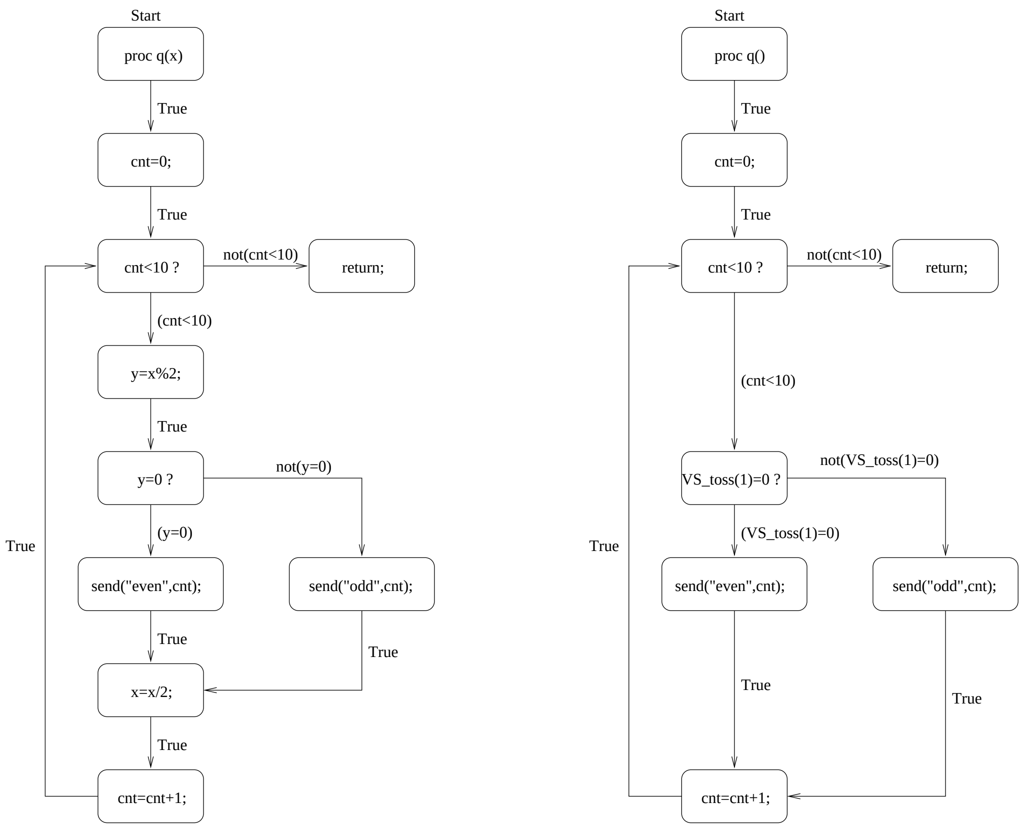

Figure 3: Another example of transformation: original G_q (left) and transformed G'_q (right)

The correctness of the algorithm is established in the next section.

5 Correctness and Precision

Let a *store* s be a function from variables (memory locations) to values. Let an *execution* $\sigma = s_0 \stackrel{n_1}{\rightarrow} s_1 \stackrel{n_2}{\rightarrow} s_2 \ldots s_{m-1} \stackrel{n_m}{\rightarrow} s_m$ of a closed system $\mathcal{S} \times E_\mathcal{S}$ be the sequence of stores s_i the system goes through while executing the sequence $n_1 \ldots n_m$ of nodes.

A variable v is said to be *functionally dependent* on an input value v_i provided by the environment $E_\mathcal{S}$ after the execution of a sequence of nodes $n_1 \ldots n_m$ of system $\mathcal{S}$ if there exist some executions σ and σ' of $\mathcal{S} \times E_\mathcal{S}$ executing the sequence of nodes $n_1 \ldots n_m$ such that σ and σ' differ only by the value v_i provided by $E_\mathcal{S}$ sometime before or during the execution and the value of v at the end of σ is different from the value of v at the end of σ'. Let $V_I[\![n_1 \ldots n_m]\!]$ denote the set of variables functionally dependent on some value provided by the environment after the execution of $n_1 \ldots n_m$.

For instance, in the following simple procedure,

```
proc p(x);
{
    a=x%2;
    b=a+1;
    c=b;
}
```

variables a, b, and c are functionally dependent on $E_\mathcal{S}$ at the end of the procedure if the value of variable x is provided by the environment. In contrast, in the following procedure,

```
proc p(x);
{
    a=0;
    if (x)
        then b=a-1;
        else b=a+1;
    c=b;
}
```

none of the variables a, b, and c are functionally dependent on the environment at the end of the procedure, even if the value of variable x is provided by the environment. Indeed, given any control-flow path leading to the end of the procedure (there are two such paths in this example), all the executions of the procedure following this path will yield the same final values for variables a, b, and c. In other words, the environment has no influence on the set of final values obtained after executing this path.

In practice, computing such sets $V_I[\![n_1 \ldots n_m]\!]$ is problematic since they are defined with respect to the executions of the system to be analyzed, whose dynamic behavior is unknown. Therefore, our algorithm rather exploits the sets $V_I(n)$, which are computed for each node of each procedure in Step 2 of the algorithm by analyzing the define-use graph of that procedure. For each node n, the set $V_I(n)$ is an upper approximation of the set of variables that are used in node n and functionally dependent of the environment after the execution of any sequence of nodes ending just before n. Formally, we have the following.

Lemma 1 *Let $V_I(n)$ denote the set of variables computed by Step 2 of the algorithm of Figure 1 for each node n in the control-flow graph G_j of a procedure p_j. Then, for any execution $\sigma = s_0 \stackrel{n_1}{\rightarrow} s_1 \stackrel{n_2}{\rightarrow} s_2 \ldots s_{m-1} \stackrel{n_m}{\rightarrow} s_m$ of the closed system $\mathcal{S} \times E_\mathcal{S}$, we have $(V_I[\![n_1 \ldots n_{m-1}]\!] \cap V(n_m)) \subseteq V_I(n_m)$.*

Proof: The proof is by induction on the length of executions. For an execution of length zero, i.e., when the system $\mathcal{S}$ is in its initial global state s_0, the first node n_1 to be executed can only be the start node of the top-level procedure of some process in the system. Since by definition start nodes do not use any variables, we have $V(n_1) = \emptyset$, and the lemma trivially holds.

Let us now prove that, if the lemma holds for executions of length smaller or equal to m, then it also holds for executions of length $m+1$. Consider an execution $s_0 \stackrel{n_1}{\rightarrow} s_1 \stackrel{n_2}{\rightarrow} s_2 \ldots s_{m-1} \stackrel{n_m}{\rightarrow} s_m \stackrel{n_{m+1}}{\rightarrow} s_{m+1}$ of $\mathcal{S} \times E_\mathcal{S}$, and a variable v in $V_I[\![n_1 \ldots n_m]\!] \cap V(n_{m+1})$. Let us show that v is in $V_I(n_{m+1})$.

If the value of v in n_{m+1} is directly defined by $E_\mathcal{S}$ sometime before the execution of node n_{m+1}, n_{m+1} is in the set $N_{E_\mathcal{S}}$ of nodes in N_j that uses the value of a variable defined by the environment $E_\mathcal{S}$ (Point 1 of Step 2). Hence, n_{m+1} is in N_I (Point 2 of Step 2). Moreover, since $v \in V(n_{m+1})$, v is in $V_I(n_{m+1})$ (Point 3 of Step 2).

Otherwise, let n_i be the last node preceding n_{m+1} in σ and σ' where v is defined. Hence, since the environment is not allowed to define variables previously defined by the system, the value of v does not change between s_i and s_{m+1}. Since $v \in V_I[\![n_1 \ldots n_m]\!]$, there exist some executions σ and σ' of $\mathcal{S} \times E_\mathcal{S}$ executing the sequence of nodes $n_1 \ldots n_m$ such that σ and σ' differ only by some input value provided by $E_\mathcal{S}$ sometime before or during the computation and the value of v at the end of σ is different from the value of v at the end of σ'. Since the value of v is the same in s_i and s_m, there also exists executions of $n_1 \ldots n_i$ that differ only by the same input value provided by $E_\mathcal{S}$ sometime before or during the computation and that lead to different values of v after the execution of n_i. Therefore, we have $v \in V_I[\![n_1 \ldots n_i]\!]$. Moreover, since v is defined in n_i, this means that there exists some other variable v' functionally dependent on $E_\mathcal{S}$ after $n_1 \ldots n_{i-1}$ that is used in n_i to define v. In other words, we have $v' \in V_I[\![n_1 \ldots n_{i-1}]\!] \cap V(n_i)$. By applying the inductive hypothesis to execution $s_0 \stackrel{n_1}{\rightarrow} s_1 \stackrel{n_2}{\rightarrow} s_2 \ldots \stackrel{n_i}{\rightarrow} s_i$, we have $v' \in V_I(n_i)$.

Consider the case where n_i corresponds to the execution of a statement in another procedure call. If n_i corresponds to an assignment statement, since $V_I(n_i) \neq \emptyset$, v is considered as being defined by $E_\mathcal{S}$ from the point of view of the procedure of n_{m+1}. Else n_i corresponds to a higher-level procedure call (fresh variables created by lower-level procedure calls are assumed not to escape their scope), and v is defined in n_i with the value of $v' \in V_I(n_i)$, then v is again considered as being defined by $E_\mathcal{S}$ from the point of view of the procedure of n_{m+1}.

Consider the case where n_i and n_{m+1} are nodes corresponding to the execution of statements of the same procedure call p_j. In this case, we know that n_i is an assignment statement defining variable v (since fresh variables created and defined by procedure calls are assumed not to escape their scope). Since $v' \in V_I(n_i)$, $V_I(n_i) \neq \emptyset$, and hence $n_i \in N_I$ (Point 3 and 4 of Step 2). Furthermore, there is an arc labeled with v from n_i to n_{m+1} in the define-use graph $\overline{G_j}$ of procedure p_j (see definition of define-use graph). Consequently, $v \in V_I(n_{m+1})$ (Point 3 of Step 2). ■

Note that Step 2 of the algorithm simply gives one way to compute approximations of the sets "$V_I[\![n_1 \ldots n_{m-1}]\!] \cap V(n_m)$" using standard notions (i.e., define-use graph) for which algorithms already exist in the literature. Of course, other algorithms could be used, provided that they can be proved to compute sets "$V_I(n)$" satisfying the previous lemma.

As we will see in Theorem 6, the following definition precisely defines the set of variables of the system $\mathcal{S}$ whose values are preserved by our algorithm.

Definition 2 Let $\sigma = s_0 \overset{n_1}{\rightarrow} s_1 \overset{n_2}{\rightarrow} s_2 \ldots s_{m-1} \overset{n_m}{\rightarrow} s_m$ be an execution of the closed system $\mathcal{S} \times E_{\mathcal{S}}$. Then, $V_{\mathcal{S}}(n_1 \ldots n_m)$ is defined inductively as follows.

- $V_{\mathcal{S}}(\epsilon) = \emptyset$.
- $V_{\mathcal{S}}(n_1 \ldots n_m)$ is defined as:
  - if n_m corresponds to an assignment statement that defines variable v and $V_I(n_m) = \emptyset$, $V_{\mathcal{S}}(n_1 \ldots n_m) = V_{\mathcal{S}}(n_1 \ldots n_{m-1}) \cup \{v\}$;
  - if n_m corresponds to an assignment statement that defines variable v and $V_I(n_m) \neq \emptyset$, $V_{\mathcal{S}}(n_1 \ldots n_m) = V_{\mathcal{S}}(n_1 \ldots n_{m-1}) \setminus \{v\}$;
  - if n_m corresponds to a procedure call and A denotes the set of fresh variables corresponding to parameters that are not removed in Point 2 of step 5 of the algorithm, $V_{\mathcal{S}}(n_1 \ldots n_m) = V_{\mathcal{S}}(n_1 \ldots n_{m-1}) \cup \{A\}$;
  - otherwise, $V_{\mathcal{S}}(n_1 \ldots n_m) = V_{\mathcal{S}}(n_1 \ldots n_{m-1})$.

■

Intuitively, $V_{\mathcal{S}}(n_1 \ldots n_m)$ reflects the accumulation of imprecisions due to the successive approximations of the sets $V_I[\![n_1 \ldots n_i]\!] \cap V(n_i)$ by $V_I(n_i)$, for all $1 \leq i \leq m$. We have the following.

Theorem 3 *Let $\sigma = s_0 \overset{n_1}{\rightarrow} s_1 \overset{n_2}{\rightarrow} s_2 \ldots s_{m-1} \overset{n_m}{\rightarrow} s_m$ be an execution of the closed system $\mathcal{S} \times E_{\mathcal{S}}$. Then, $V_I[\![n_1 \ldots n_m]\!] \cap V_{\mathcal{S}}(n_1 \ldots n_m) = \emptyset$.*

Proof: Follows from Lemma 1 by an induction on the length of executions. ■

When, for any execution $\sigma = s_0 \overset{n_1}{\rightarrow} s_1 \overset{n_2}{\rightarrow} s_2 \ldots s_{m-1} \overset{n_m}{\rightarrow} s_m$ of the closed system $\mathcal{S} \times E_{\mathcal{S}}$, every variable defined in s_m is either in $V_I[\![n_1 \ldots n_m]\!]$ or in $V_{\mathcal{S}}(n_1 \ldots n_m)$, the approximations due to the sets $V_I(n_i)$, $1 \leq i \leq m$, are optimal. We will come back to this point when we will discuss the precision of our algorithm at the end of this section.

A property similar to Lemma 1 holds for $V_{\mathcal{S}}$.

Lemma 4 *Let $\sigma = s_0 \overset{n_1}{\rightarrow} s_1 \overset{n_2}{\rightarrow} s_2 \ldots s_{m-1} \overset{n_m}{\rightarrow} s_m$ be an execution of the closed system $\mathcal{S} \times E_{\mathcal{S}}$. Then, we have $(V(n_m) \setminus V_I(n_m)) \subseteq V_{\mathcal{S}}(n_1 \ldots n_{m-1})$.*

Proof: Let v be a variable in $V(n_m)$ that is not in $V_I(n_m)$. Let us show that $v \in V_{\mathcal{S}}(n_1 \ldots n_{m-1})$. Since $v \notin V_I(n_m)$, we know v is not defined by $E_{\mathcal{S}}$. Hence, let n_i denote the last node in $n_1 \ldots n_m$ where v has been defined. Consider the case where n_i corresponds to the execution of a statement in another procedure call. If n_i corresponds to an assignment statement, then $V_I(n_i) = \emptyset$, otherwise v would be considered as defined by the environment from the point of view of the procedure of n_m. Else n_i corresponds to a higher-level procedure call (fresh variables created by lower-level procedure calls are assumed not to escape their scope), and v is a fresh variable corresponding to a parameter not removed in Point 2 of Step 5 of the algorithm, since v would be otherwise considered as defined by the environment from the point of view of the procedure of n_m. Therefore, by Definition 2, v was added to $V_{\mathcal{S}}(n_1 \ldots n_i)$ in n_i. Consider now the other case where n_i and n_m are nodes corresponding to the execution of statements in the same procedure call. Then, n_i corresponds to an assignment statement (since fresh variables created and defined by procedure calls are assumed not to escape their scope), and there is a define-use arc from n_i to n_m labeled with v (see definition of define-use graph); this implies that $V_I(n_i) = \emptyset$, since otherwise, by Point 3 of Step 2 of the algorithm, $V_I(n_m)$ would contain v. Therefore, by Definition 2, v was added to $V_{\mathcal{S}}(n_1 \ldots n_i)$ in n_i.

Since n_i is the last node in $n_1 \ldots n_m$ where v has been defined during the execution of $n_i \ldots n_m$, v is in $V_{\mathcal{S}}(n_1 \ldots n_m)$. ■

We now prove that the system $\mathcal{S}'$ obtained after the transformation is closed.

Lemma 5 *For any node n' in the control-flow graph G'_j of any procedure p'_j generated by the algorithm of Figure 1, we have $V_I(n') = \emptyset$.*

Proof: Each node n' of a control-flow graph G'_j either (1) is a conditional statement testing the value returned after a call to VS_toss, introduced in Step 4 of the algorithm, or (2) corresponds to a marked node n of the original control-flow graph G_j. For nodes of type (1), the lemma trivially holds.

Let us now consider nodes of type (2). By Step 3 of the algorithm, we know that the corresponding node n of G_j is associated with either (2.1) a start node, (2.2) a termination statement, (2.3) a procedure call, or (2.4) an assignment or conditional statement for which $V_I(n) = \emptyset$. Since start nodes and nodes corresponding terminal statements do not use any variables, we have $V(n) = \emptyset$ and hence $V_I(n) = \emptyset$, i.e., the lemma holds for subcases (2.1) and (2.2). By Step 5 of the algorithm, references to variables in $V_I(n)$ are eliminated. Therefore, the lemma holds for subcases (2.3) and (2.4). ■

Since we have proved that the system $\mathcal{S}'$ obtained after the transformation performed by our algorithm is closed and hence self-executable, we can now consider its executions and relate them precisely to the executions of $\mathcal{S}$ in conjunction with $E_{\mathcal{S}}$.

Specifically, the following theorem establishes a formal correspondence between *computations* of both systems. Let π denote a finite sequence of nodes $n_0 n_1 \ldots n_k$ executed by a single process such that n_0 is a node marked in Step 3 of the algorithm, and, for all $1 \leq j \leq k$, n_j is unmarked. A *computation* $\sigma = s_0 \overset{\pi_1}{\rightarrow} s_1 \overset{\pi_2}{\rightarrow} s_2 \ldots s_{m-1} \overset{\pi_m}{\rightarrow} s_m$ of a closed system is then defined as the finite sequence of stores s_i the system goes through while executing the sequences π_i of nodes.

In what follows, all the nodes in G'_j that are added by Point 1 of Step 4 of the algorithm are considered as marked in G'_j, while all the other nodes of G'_j are considered as unmarked. This implies that there is a one-to-one correspondence between the marked nodes n and n' of G_j and G'_j respectively. We write $n = n'$ to denote this correspondence. By construction (see algorithm), the statements associated with corresponding marked nodes are either identical or are both procedure calls that may differ only by their arguments.

We write $v \in s$ to denote a variable that is defined in store s. Let $s(v)$ denote the value of variable v defined in store s. Let π^0 denote the first node n_0 of the sequence of nodes $\pi = n_0 n_1 \dots n_k$. We write $\pi_i^0 \equiv {\pi'}_i^0$ to mean that $\pi_i^0 = {\pi'}_i^0$ and that these nodes are executed by corresponding processes P and P' of $\mathcal{S}$ and $\mathcal{S}'$ respectively. Let $next(s)$ denote the set of marked nodes next to be executed from a store s in a computation. Since there is exactly one such node per process (which may possibly be blocking), the size of $next(s)$ is constant and equal to the number of processes in the system. We write $next(s_i) \equiv next(s'_i)$ to mean that $\forall n \in next(s_i) : \exists n' \in next(s'_i) : n \equiv n'$ and $\forall n' \in next(s'_i) : \exists n \in next(s_i) : n' \equiv n$.

We are now ready to state the main theorem that defines the correctness of our algorithm.

Theorem 6 *Let $\mathcal{S}$ be an open system that satisfies all the assumptions previously defined and implemented by a set of procedures p_j. Let $\mathcal{S}'$ be the system implemented by the set of procedures p'_j obtained by transforming each procedure p_j using the algorithm of Figure 1. Then, for every computation $\sigma = s_0 \xrightarrow{\pi_1} s_1 \xrightarrow{\pi_2} s_2 \dots s_{m-1} \xrightarrow{\pi_m} s_m$ of $\mathcal{S} \times E_{\mathcal{S}}$, there exists a computation $\sigma' = s'_0 \xrightarrow{\pi'_1} s'_1 \xrightarrow{\pi'_2} s'_2 \dots s'_{m-1} \xrightarrow{\pi'_m} s'_m$ of $\mathcal{S}'$ such that, $\forall 0 \leq i \leq m$, the three following properties are satisfied:*

1. *if $i > 0$, $\pi_i^0 \equiv {\pi'}_i^0$;*
2. *$next(s_i) \equiv next(s'_i)$; and*
3. *$\forall v \in V_{\mathcal{S}}(\pi_1 \dots \pi_i) : s_i(v) = s'_i(v)$.*

Proof: The proof is by induction on the length of computations. For computations of length zero, the closed systems $\mathcal{S} \times E_{\mathcal{S}}$ and $\mathcal{S}'$ are in their initial global states s_0 and s'_0 respectively, and Property 1 is vacuously true. The set $next(s_0)$ contains the start node of the top-level procedure executed by each process P in $\mathcal{P}$ of the system $\mathcal{S}$. Since start nodes are preserved by the algorithm (Point 1 of Step 3), we have $next(s_0) \equiv next(s'_0)$. Since initially, we assume no variables have been defined by the system yet, we have $V_{\mathcal{S}}(\epsilon) = \emptyset$, and Property 3 of the theorem trivially holds.

Let us consider a computation σ_m of $\mathcal{S} \times E_{\mathcal{S}}$ of length m and the corresponding computation σ'_m of $\mathcal{S}'$ obtained by the induction hypothesis. Let us show that, for any possible computation σ_{m+1} of length $m+1$ extending σ_m by a sequence $\pi_{m+1} = n_0 n_1 \dots n_k$ of nodes, there exists a sequence $\pi'_{m+1} = n'_0 n'_1 \dots n'_{k'}$ of nodes extending σ'_m to a computation σ'_{m+1} such that $\pi_{m+1}^0 \equiv {\pi'}_{m+1}^0$, $next(s_{m+1}) \equiv next(s'_{m+1})$ and $\forall v \in V_{\mathcal{S}}(\pi_1 \dots \pi_{m+1}) : s_{m+1}(v) = s'_{m+1}(v)$.

Let P denote the process of $\mathcal{S}$ which executes the sequence of nodes $n_0 n_1 \dots n_k$. Let P' be the process of $\mathcal{S}'$ corresponding to P. Since n_0 is a node of some procedure p_j that is marked by Step 3 of the algorithm, let n'_0 be the corresponding node of G'_j such that $n_0 = n'_0$. Since σ_{m+1} is a computation, n_0 is not blocking in s_m. Since $n_0 \in next(s_m)$, it follows from the inductive hypothesis that $n'_0 \in next(s'_m)$. If n_0 is not an operation on a communication object, then neither is n'_0 and hence n'_0 is also not blocking. Otherwise, we know, by assumption, that the enabledness of any operation on any communication object depends exclusively on the sequence of operations that has been performed on the object in the history of the system, and not on the values that are possibly stored or passed through the object, or passed as an argument to the operation. Since all operations on communication objects are procedure calls, the corresponding nodes are marked in the calling procedures and in their transformed versions. Therefore, from the inductive hypothesis, it is easy to show that the projections of σ_m and σ'_m onto nodes corresponding to operations on any given communication object are identical. Since $n_0 = n'_0$ and n_0 is not blocking in s_m, n'_0 is not blocking in s'_m. Thus, we have $\pi_{m+1}^0 \equiv {\pi'}_{m+1}^0$.

By definition of a computation, we know that the next node to be executed by P from s_{m+1} is a marked node, let us denote it by n_{k+1}. Hence, we have $next(s_{m+1}) = (next(s_m) \setminus \{n_0\}) \cup \{n_{k+1}\}$. Let n'_{k+1} be the marked node for $\mathcal{S}'$ corresponding to the node n_{k+1}. In order to prove that $next(s'_{m+1}) = (next(s'_m) \setminus \{n'_0\}) \cup \{n'_{k+1}\}$ and hence that $next(s_{m+1}) \equiv next(s'_{m+1})$, we have to show that there is an execution of process P from s'_m that leads to n'_{k+1} without traversing any marked nodes other than n'_0. Two cases are possible: either n_0 and n_{k+1} correspond to the executions of statements of the same procedure call p_j, or they do not. We consider these two cases successively.

If n_0 and n_{k+1} are nodes of procedure p_j, then n'_0 and n'_{k+1} are nodes of procedure p'_j. This also means that the marked node n_0 corresponds to either a start node, or an assignment statement, or a conditional statement, and that $V_I(n_0) = \emptyset$. Therefore, there is exactly one arc a from n_0 in G_j whose label evaluates to true in s_m. This arc leads to node n_1. Let $succ(a)$ denote the set of marked nodes of G_j that are reachable from n_0 by a sequence aw of control-flow arcs in G_j passing through unmarked nodes exclusively and starting with arc a. Since $n_{k+1} \in succ(a)$, $|succ(a)| \geq 1$. Therefore, by Point 2 of Step 4 of the algorithm, there is an arc a' from n'_0 in G'_j that is labeled with the boolean expression labeling arc a. Since $V_I(n_0) = \emptyset$, we know by Lemma 4 that $V(n_0) \subseteq V_{\mathcal{S}}(\pi_1 \dots \pi_m)$. Therefore, by applying Property 3 of the inductive hypothesis, we have $\forall v \in V_{\mathcal{S}}(\pi_1 \dots \pi_m) : s_m(v) = s'_m(v)$, and thus $\forall v \in V(n_0) : s_m(v) = s'_m(v)$. By definition of a control-flow graph, all variables occurring in the boolean expression labeling arc a are in $V(n_0)$. Thus, if the boolean expression labeling arc a evaluates to true in s_m, then the same boolean expression, which labels arc a', also evaluates to true in s'_m. Since this boolean expression labels arc a' in G'_j, the execution of node n'_0 from s'_m leads to the successor node of arc a' in G'_j, let us call it n'_1.

If $|succ(a)| = 1$, by Point 2.2 of Step 4 of the algorithm, n'_1 is the node corresponding to n_{k+1}, i.e., n'_{k+1}, which concludes the proof of this case. Otherwise, we have $|succ(a)| > 1$. By Point 2.3 of Step 4 of the algorithm, node n'_1 corresponds to a test on the value returned by a call to the nondeterministic function VS_toss, and there is an arc from n'_1 to n'_{k+1} in G'_j. Since boolean expressions testing the value returned by a call to VS_toss introduced in Step 4 always evaluate to true in any store, by taking $\pi_{m+1} = n'_0 n'_1$, we define an execution of process P from s'_m that leads to n'_{k+1}.

We now consider the case where n_0 and n_{k+1} correspond to the executions of statements of two different procedure calls p_j and p_l. (Note that, in case of a recursive procedure call, p_l executes the same code as p_j.) This implies that n'_0 and n'_{k+1} also correspond to the executions of statements of two different procedure calls p'_j and p'_l. Two cases are possible: n_0 corresponds either to a procedure call, or to a termination statement. If n_0 is a procedure call, n_{k+1} is the start node of the procedure p_l being called. Since procedure calls and start nodes are marked nodes, the execution of n'_0 leads directly to n'_{k+1}, which completes the proof of Property 2 for this case. Otherwise, n_0 is a termination statement. Since n_0 is not blocking in s_m, p_j is not the top-level procedure call of this process, and procedure p_l is the procedure that called p_j. Thus n_1 must be the successor node in G_l of the node of G_l that called p_j (by construction, every node corresponding to a procedure call has always exactly one successor node in a control-flow graph). Let a be the arc from this node to n_1 in G_l. The rest of the proof for this case is identical to the case analysis on $|succ(a)|$ done in the previous paragraph. This concludes the proof of Property 2.

We now prove Property 3. Recall that the environment cannot define any variables previously defined by the system and hence any variables in $V_{\mathcal{S}}(\pi_1 \ldots \pi_m)$. The key observation to prove Property 3 is then that, by Definition 2, for any node n that does not correspond to an assignment statement with $V_I(n) = \emptyset$, or does not correspond to a procedure call with a nonempty set A of fresh variables corresponding to parameters not removed in Point 2 of step 5 of the algorithm, we have $V_{\mathcal{S}}(\pi_1 \ldots \pi_m) \supseteq V_{\mathcal{S}}(\pi_1 \ldots \pi_m n_0)$.

Let s and s' denote the stores reached by processes P and P' after the execution of nodes n_0 and n'_0 from stores s_m and s'_m, respectively. If node n_0 satisfies the conditions stated in the previous paragraph, we immediately have $\forall v \in V_{\mathcal{S}}(\pi_1 \ldots \pi_m n_0) : s(v) = s'(v)$.

If the marked node n_0 corresponds to an assignment statement that defines a variable v_0, we know $n_0 \notin N_I$ (Point 4 of Step 3 of the algorithm), and $V_I(n_0) = \emptyset$. By Lemma 4, this means that $V(n_0) \subseteq V_{\mathcal{S}}(\pi_1 \ldots \pi_m)$. Therefore, by applying Property 3 of the inductive hypothesis, we have $\forall v \in V_{\mathcal{S}}(\pi_1 \ldots \pi_m) : s_m(v) = s'_m(v)$, and thus $\forall v \in V(n_0) : s_m(v) = s'_m(v)$. This implies that n_0 and n'_0 performs identical store transformations. Therefore, v_0 is defined both in s and s' with the same value. Moreover, by Definition 2, $V_{\mathcal{S}}(\pi_1 \ldots \pi_m n_0) = V_{\mathcal{S}}(\pi_1 \ldots \pi_m) \cup \{v_0\}$. We thus have $\forall v \in V_{\mathcal{S}}(\pi_1 \ldots \pi_m n_0) : s(v) = s'(v)$.

Consider the case where the marked node n_0 corresponds to a procedure call with a nonempty set A of fresh variables corresponding to parameters not removed in Point 2 of step 5 of the algorithm. Let A' be the set of variables whose values are passed as argument via such parameters, and hence copied into some fresh variable in A. Hence, we have $A' \subseteq V(n_0)$ and $A' \cap V_I(n_0) = \emptyset$. By Lemma 4, we know $(V(n_0) \setminus V_I(n_0)) \subseteq V_{\mathcal{S}}(\pi_1 \ldots \pi_m)$. Therefore, $A' \subseteq V_{\mathcal{S}}(\pi_1 \ldots \pi_m)$. Therefore, by applying Property 3 of the inductive hypothesis, we have $\forall v \in V_{\mathcal{S}}(\pi_1 \ldots \pi_m) : s_m(v) = s'_m(v)$, and thus $\forall v \in A' : s_m(v) = s'_m(v)$. This implies that $\forall v \in A : s(v) = s'(v)$. By Definition 2, $V_{\mathcal{S}}(\pi_1 \ldots \pi_m n_0) = V_{\mathcal{S}}(\pi_1 \ldots \pi_m) \cup \{A\}$. We thus have again $\forall v \in V_{\mathcal{S}}(\pi_1 \ldots \pi_m n_0) : s(v) = s'(v)$.

By repeating the same argument for the remaining nodes in π_{m+1} (if any), the proof of Property 3 is complete. ■

Intuitively, the previous theorem states that, for every computation σ of $\mathcal{S} \times E_{\mathcal{S}}$, the "projection" of σ (including call stacks of processes) onto the set of variables that do not use variables defined by or functionally dependent on $E_{\mathcal{S}}$ is preserved by the transformation performed by our algorithm.

Note that the original correctness criterion used in Theorem 6 combines aspects of both reactive and functional program semantics: Points 1 and 2 of the theorem establishes a *simulation relation* between $\mathcal{S} \times E_{\mathcal{S}}$ and $\mathcal{S}'$, while Point 3 of the theorem establishes a *functional equivalence* for a subset of the values computed by $\mathcal{S} \times E_{\mathcal{S}}$ and $\mathcal{S}'$.

From Theorem 6, it is then easy to show that deadlocks and assertion violations that do not test expressions involving values provided by the environment $E_{\mathcal{S}}$ are preserved by the transformation performed by the algorithm. Precisely, recall that assertions are visible operations, and hence procedure calls from the point of view of our algorithm. To be consistent with our previous assumption that only variables can be passed as arguments to procedure calls, we assume every assertion in $\mathcal{S}$ has exactly one argument, which is a variable whose value determines whether or not the assertion is violated. We say that an assertion that corresponds to some (by construction, marked) node n in $\mathcal{S}$ is *preserved in* $\mathcal{S}'$ if the variable passed as argument in n is not eliminated by Point 2 of Step 5 of the algorithm of Figure 1 in the corresponding node n' in $\mathcal{S}'$. We then have the following.

Theorem 7 *Let $\mathcal{S}$ and $\mathcal{S}'$ be defined as in Theorem 6. Let $A_{\mathcal{S} \times E_{\mathcal{S}}}$ denote the state space of the closed system $\mathcal{S} \times E_{\mathcal{S}}$ obtained by combining $\mathcal{S}$ with its most general environment $E_{\mathcal{S}}$, and let $A_{\mathcal{S}'}$ denote the state space of the closed system $\mathcal{S}'$. Then, all the deadlocks in $A_{\mathcal{S} \times E_{\mathcal{S}}}$ are in $A_{\mathcal{S}'}$. Moreover, for all the assertions in procedures p_j preserved in p'_j, if there exists a global state in $A_{\mathcal{S} \times E_{\mathcal{S}}}$ where such an assertion is violated, then there exists a global state in $A_{\mathcal{S}'}$ where the same assertion is violated.*

Proof: Consider a deadlock s in $A_{\mathcal{S} \times E_{\mathcal{S}}}$. By definition, a deadlock is a reachable global state in $A_{\mathcal{S} \times E_{\mathcal{S}}}$ where all the processes are blocked. Let σ be a computation of $\mathcal{S} \times E_{\mathcal{S}}$ that leads to the deadlock. By applying Theorem 6, we know that there exists a computation σ' of $\mathcal{S}'$ that leads to a global state s' such that $next(s) \equiv next(s')$. Since the executions of all the nodes in $next(s)$ are blocking, all the nodes in $next(s)$ attempt to perform a visible operation. Since the enabledness of any operation on any communication object depends exclusively on the sequence of operations that has been performed on the object in the history of the system, and since this history is the same in σ and σ' for all communication objects by Property 1 of Theorem 6, the executions of all the nodes in $next(s')$ are also blocking, and s' is a deadlock in $A_{\mathcal{S}'}$.

Let us now consider the case of assertion violations. Let s be a global state s in $A_{\mathcal{S} \times E_{\mathcal{S}}}$ where an assertion is violated. This means that there is a node n in $next(s)$ corresponding to this assertion which tests the value of a variable v. Hence, $v \in V(n)$. Since the assertion is violated, the value of v in s is "false". Let σ be a computation of $\mathcal{S} \times E_{\mathcal{S}}$ that leads to s. By applying Theorem 6, we know that there exists a computation σ' executing $\pi_1 \ldots \pi_m$ of $\mathcal{S}'$ that leads to a global state s' such that $next(s) \equiv next(s')$. Let n' be the node in $next(s')$ such that $n \equiv n'$. Since the assertion is preserved in $\mathcal{S}'$, we know that $V_I(n) = \emptyset$, and that n' also tests the value of variable v. By Lemma 4, $V_I(n) = \emptyset$ implies that $V(n) \subseteq V_{\mathcal{S}}(\pi_1 \ldots \pi_m)$. Since $v \in V(n)$, by Property 3 of Theorem 6, we conclude that $s(v) = s'(v)$. Hence, the assertion is also violated in s' in $A_{\mathcal{S}'}$. ■

Note that the transformation from $\mathcal{S}$ to $\mathcal{S}'$ eliminates some program statements from $\mathcal{S}$, yet it must preserve all possible behaviors of $\mathcal{S}$ as specified above. One must be careful to ensure that when the transformation removes program statements from $\mathcal{S}$ that may lead to run-time errors, these errors cannot generate extra executions in $\mathcal{S}$ (that $\mathcal{S}'$ would thus not preserve). Therefore, we make the following observations about the run-time errors in the original system $\mathcal{S}$. If the semantics of the source language specifies the behavior of a particular run-time error, that behavior needs to be preserved in $\mathcal{S}'$ if it may lead to additional executions in $\mathcal{S}$. Otherwise, for all errors whose behavior is not specified by the source-language semantics, but rather left to the particular implementation of the language, $\mathcal{S}'$ does not need to preserve any particular choice of behavior. In other words, the correctness of the transformation is defined relative to the source language itself rather than a particular implementation of the source language. For example, C does not specify the behavior of run-time errors such as array-out-of-bounds, and so the transformation algorithm for C programs may freely remove array references when appropriate. In contrast, an array-out-of-bounds error in an ML program throws an exception, and so $\mathcal{S}'$ needs to preserve any bounds check whose exception may lead to additional executions in $\mathcal{S}$.

An *optimal* translation of an open system $\mathcal{S}$ is a closed system $\mathcal{S}'_{\rm opt}$ such that the properties in Theorem 6 additionally hold in the other direction. In other words, for every computation of $\mathcal{S}'_{\rm opt}$ there exists a computation in $\mathcal{S} \times E_{\mathcal{S}}$ that satisfies the three properties in the theorem. For example, consider Figure 3 and consider a system $\mathcal{S}$ that comprises only the procedure q(x). Then the algorithm performs an optimal translation as explained in the text that accompanies that example. In contrast, the translation in Figure 2 is not optimal.

Of course, it is not possible in general to achieve an optimal translation. There are several sources of conservative approximation in our algorithm.

Interprocedural issues: The algorithm assumes that it is known which nodes in a control-flow graph of procedure p use the value of a variable defined by the environment $E_{\mathcal{S}}$. The source of this information may be manual, in the form of a specification, or automatic, in the form of an interprocedural analysis on top of our intraprocedural analysis. If manual, then it is only possible to achieve an optimal specification if the entire open system $\mathcal{S}$ is known in advance; otherwise, it will be necessary to assume conservatively that any input variables or variables whose addresses escape are defined by the environment. If automatic, then the interprocedural analysis will necessarily be approximate in general due to well-known sources of imprecision such as escaping variables, the call-context problem, and so forth.

Dataflow analysis: There are the standard imprecisions associated with a classic dataflow analysis. For instance, any may-alias analysis may generate spurious dependencies. Furthermore, composing define-use arcs is imprecise. For example, the code `a=x+1; b=a-x;` will report incorrectly that `b` is dependent upon `x`. Lemma 1 covers this source of imprecision.

Finite variance: In a given run of a system, some executions of a given node may be functionally dependent on $E_{\mathcal{S}}$ and others not. Our algorithm conservatively removes such nodes. In brief, any dataflow analysis must be of finite variance and thus may suffer this kind of imprecision. Our analysis is monovariant, and the corresponding imprecision is summarized in Theorem 3.

Temporal independence: In a given run of a system, there may be multiple executions of a given conditional node that are all functionally dependent on $E_{\mathcal{S}}$ in precisely the same way. For instance, consider Figure 2. The open program encounters the conditional test 10 times per call to p, but goes down the same branch each time. The particular branch is dependent on $E_{\mathcal{S}}$. The translated closed program performs 10 VS_toss operations rather than a single one before the loop. In this case, hoisting the conditional test "y=0" outside the loop in p would have eliminated this imprecision. In general, however, it is unavoidable.

Note that the above discussion concerns only the precision of our algorithm with respect to the possible sets of computations satisfying Theorem 6. One can also discuss the optimality of the branching structure of the generated program. For instance, sequences of VS_toss that result in the same sequences of marked nodes are redundant, and could thus be eliminated. This line of thought will not be discussed further here.

6 Applications

The motivation behind this work is the desire to analyze automatically very large concurrent reactive systems for which reliability is critical. One such system is Lucent Technologies' 5ESS telephone switching system [MS85], which provides telecommunications services for land-line and wireless networks. The 5ESS software consists of thousands of interacting concurrent reactive processes, and is comprised of millions of lines of code, mostly written in the C programming language. The software is continually evolving as new features are added.

The sheer size, complexity, and changing nature of the code renders it extremely difficult to understand the possible interactions between the processes in the switch. Such interactions are often extremely hard to reproduce and analyze in the existing testing environments available in the 5ESS development organization. These include an on-line simulation environment, which can execute a complete version of the 5ESS software, and testing labs where the 5ESS software can be executed on actual 5ESS hardware switches. A developer using either of these testing environments must first set a significant number of data configurations before being able to run a test. Non-determinism among concurrent processes is implicitly resolved by the execution environment. Therefore, it can be difficult to reproduce a specific scenario.

Recently, we have started investigating in collaboration with a group of 5ESS developers how the techniques introduced in this paper combined with VeriSoft could be used for providing a new "lightweight" testing and reverse-engineering platform for reactive properties of 5ESS code. As a case study, we considered a large multi-process 5ESS application that is responsible for providing call processing features – such as originations, terminations, location registration, hand over, roaming, and call forwarding – for specific wireless systems. This call processing software describes about 10 main families of concurrent reactive processes. The code describing each family of processes ranges from approximately 30,000 to several hundred thousand lines of C code.

In order to be able to execute this code for analyzing its dynamic behavior without using the existing heavyweight execution environments, we first needed a way to make the application "stand-alone". Therefore, we implemented the algorithm described in this paper in a prototype tool for automatically closing open programs written in the C programming language. We manually developed software stubs for providing a small number of inputs corresponding to basic external events we wanted to control in order to trigger, and observe afterwards, interactions between concurrent processes of the application. The remainder of the system was closed automatically using our tool. At the time of this writing, VeriSoft is currently being used to analyze the dynamic behavior of the closed application. Note that completely closing this application by hand is clearly impractical because it would require developing and maintaining code for simulating a substantial portion of the entire 5ESS switch software and databases.

7 Related Work and Conclusions

The core of our algorithm for closing an open system is a dataflow analysis of C or C-like procedures. The dataflow analysis is phrased as a *graph-reachability* problem. This is a common approach to dataflow problems that our algorithm shares with many others, such as [Cal88, CK88, Kou77], to name but a few. In particular, we use standard techniques for computing define-use dependencies, such as [ASU86, FOW87, MR90]. These techniques rely on a (conservative) solution to the aliasing problem. Examples of alias analyses include [CWZ90, Deu94, Lan91, Ruf95, SRW96], to name a few of many.

Using graph-reachability-based dataflow analyses to drive transformations of imperative programs is a well established technique. For instance, [HR92] describes transformations based on the Program Dependence Graph [FOW87, KKL+81]. Perhaps the most common such transformation is *program slicing*, originally introduced by Weiser in [Wei81] and later much investigated and extended (e.g., [HRB88, Tip95]). The input to a typical slicing tool is a program, a point p within the program, and an identifier x. The output is a possibly reduced program that preserves the trace of values bound to x at p. Note that a correct (albeit useless) slicing algorithm is the identity transformation. Our transformation is different in that it *must* eliminate some parts of the original program — namely, the parts that depend on values supplied by the environment. Our transformation may seem similar to a simultaneous slice on each identifier at each program point that does *not* depend on a value supplied by the environment the environment, where those identifiers are pre-computed by a forward analysis. However, such a slice would be too large. For instance, consider the example programs in Figures 2 and 3. In each case, the trace of *cnt* values depends on the conditional test involving y; hence, the above slicing procedure would not eliminate the the nodes involving y, and the resulting program would remain open. To close the program, we must eliminate *all* nodes that depend on a value provided by the environment. Therefore, in contrast to slicing, we require only *inclusion* rather than equivalence of executions, and we are forced to introduce VS_toss in order to achieve in a single transformation both inclusion of executions and complete elimination of nodes dependent upon the environment.

Combining an open system with its most general environment is related to the idea of *hiding* a set of visible actions of a process [Hoa85, Mil89] in a process calculus. The originality and technical contributions of our work are to apply this idea to full-fledged programming languages, instead of simple transition systems.

Systematically exploring the state space of the implementation of a concurrent system written in a programming language such as C, rather than constructing and/or analyzing an approximate model of that implementation, is a new approach to concurrent program analysis. This work continues the line of research set forth in [God97]. A complementary approach to analyzing such systems is to use static analysis, such as *abstract interpretation* [CC77]. Most work has involved analyzing the communication patterns that occur in a system [Tay83, LC91, MR93, Col95, Cri95, Ven97]. A model checker could analyze the results of such static analyses in order to prove the absence of certain specific types of errors. In contrast, our approach is based on *dynamic* observation of a system. This opens up the possibility of detecting a wider range of behaviors that may have been abstracted away by a static analysis. In this paper, we have shown that this kind of dynamic analysis can be used even on open systems by first applying a static analysis — not to construct a model of the system, but rather to transform it to a closed form.

Our algorithm is a first solution to the general problem of closing an open system. There is evidence to suggest that this algorithm can be applied to large pieces of code. The algorithm also has the significant practical benefit that it can close *any* open system completely automatically. It is also applicable to open sequential systems, i.e., systems comprising only a single process.

The experience that we have gained in developing this transformation algorithm has shed some light on possible ways to improve the precision of the result. Consider, for instance, a resource-management system that receives (via its open interface) 32-bit integers representing amounts of time requested from the resource, but whose visible behavior only depends on which of a small set of ranges each request falls into. Our transformation would completely eliminate the open interface in order to avoid the intractability that arises from the systematic exploration of all possible inputs. However, one could hope for a static analysis that would determine the appropriate partitioning of the input domain, and, if it is small enough, *simplify* the interface instead of eliminating it. Consider further that the original system contains a control path along which are two conditional tests that both depend on an input, say the time request, but always evaluate to the same value. The current algorithm inserts a VS_toss operation at both points. A static analysis that could detect this property could cut the possible branching of the state-space exploration in half. Both of these examples are special cases of the general problem of *symbolically* analyzing the behavior of an imperative program with respect to unknown values (the open interface). We are investigating the applicability of an existing symbolic analysis [Col96] to this problem.

References

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Princi-*

ples, Techniques and Tools. Addison-Wesley, 1986.

[Cal88] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the ACM '88 Conference on Programming Lanugage Design and Implementation*, pages 47–56, Atlanta, GE, USA, June 1988. ACM Press.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles*, pages 238–252, New York, NY, 1977. ACM.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.

[CK88] K. D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In David S. Wise, editor, *Proceedings of the SIGPLAN '88 Conference on Programming Lanugage Design and Implementation (SIGPLAN '88)*, pages 57–66, Atlanta, GE, USA, June 1988. ACM Press.

[Col95] C. Colby. Analyzing the communication topology of concurrent programs. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 202–213, New York, NY, USA, June 1995. ACM Press.

[Col96] C. Colby. *Semantics-based Program Analysis via Symbolic Composition of Transfer Relations.* PhD thesis, Carnegie Mellon University, August 1996.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 1(15):36–72, 1993.

[Cri95] R. Cridlig. Semantic analysis of shared-memory concurrent languages using abstract model-checking. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 214–225, New York, NY, USA, June 1995. ACM Press.

[CWZ90] D.R. Chase, M. Wegman, and F.K. Zadeck. Analysis of pointers and structures. In *Proccedings of Conference on Programming Language Design and Implementation*, pages 296–310, June 1990.

[Deu94] A. Deutsch. Interprocedural May-Alias analysis for pointers: Beyond k-limiting. *SIGPLAN Notices*, 29(6):230–241, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[God96] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science.* Springer-Verlag, January 1996.

[God97] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, Paris, January 1997.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[HR92] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *Proceedings of the 14th International Conference on Software Engineering*, pages 392–411, May 1992.

[HRB88] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In David S. Wise, editor, *Proceedings of the ACM '88 Conference on Programming Language Design and Implementation*, pages 35–46, Atlanta, GE, USA, June 1988. ACM Press.

[KKL+81] D. J. Kuck, R. H. Kuhn, B. Leasure, D. A. Padua, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[Kou77] L. T. Kou. On live-dead analysis for global data flow problems. *Journal of the ACM*, 23(3):473–483, July 1977.

[Lan91] W. Landi. *Interprocedural aliasing in the presence of pointers.* PhD thesis, Rutgers University, 1991.

[LC91] D. L. Long and L. A. Clarke. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *Proceedings of ACM Symposium on Testing, Analysis, and verification (TAV4)*, pages 21–35, Vancouver, October 1991.

[McM93] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[MR90] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks. *Acta Informatica*, 28:121–163, 1990.

[MR93] S. P. Masticola and B. G. Ryder. Non-concurrency analysis. In *Proceedings of Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel programming*, pages 129–138, San Diego, May 1993.

[MS85] K.E. Martersteck and A.E. Spencer. Introduction to the 5ESS(TM) switching system. *AT&T Technical Journal*, 64(6 part 2):1305–1314, July-August 1985.

[QS81] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int'l Symp. on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1981.

[Ruf95] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of Conference on Programming Language Design and Implementation*, New York, NY, USA, June 1995. ACM Press.

[SRW96] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 16–31, St. Petersburg, Florida, January 1996. ACM Press.

[Tay83] R. N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, pages 362–376, May 1983.

[Tip95] F. Tip. Generic techniques for source-level debugging and dynamic program slicing. In *Proceedings of TAPSOFT'95*, volume 915 of *Lecture Notes in Computer Science.* Springer-Verlag, 1995.

[Ven97] A. Venet. Abstract interpretation of the π-calculus. In Mads Dam, editor, *Analysis and Verification of Multiple-Agent Languages (Proceedings of the Fifth LOMAPS Workshop)*, volume 1192 of *Lecture Notes in Computer Science*, pages 51–75. Springer-Verlag, 1997.

[Wei81] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, March 1981.