TOWARDS SCALABLE COMPOSITIONAL ANALYSIS[*]

James C. Corbett[†] and George S. Avrunin

CMPSCI Technical Report 94–56
July 1994

Software Development Laboratory
Computer Science Department
Lederle Graduate Research Center
University of Massachusetts
Amherst, Massachusetts 01003-4601

[*]This paper has been submitted for publication. The research reported here was partially supported by grants from the National Science Foundation and the Office of Naval Research.

[†]Information and Computer Science Department, University of Hawaii at Manoa, Honolulu, HI 96822

TOWARDS SCALABLE COMPOSITIONAL ANALYSIS

JAMES C. CORBETT AND GEORGE S. AVRUNIN

Abstract. Due to the state explosion problem, analysis of large concurrent programs will undoubtedly require compositional techniques. Existing compositional techniques are based on the idea of replacing complex subsystems with simpler processes with the same interfaces to their environments, and using the simpler processes to analyze the full system. Most algorithms for proving equivalence between two processes, however, require enumerating the states of both processes. When part of a concurrent system consists of many highly coupled processes, it may not be possible to decompose the system into components that are both small enough to enumerate and have simple interfaces with their environments. In such cases, analysis of the systems by standard methods will be infeasible.

In this paper, we describe a technique for proving trace equivalence of *deterministic* and *divergence-free* systems without enumerating their states. (For deterministic systems, essentially all the standard notions of process equivalence collapse to trace equivalence, so this technique also establishes failures equivalence, observational equivalence, etc.) Our approach is to generate necessary conditions for the existence of a trace of one system that is not a trace of the other; if the necessary conditions cannot be satisfied the systems are equivalent. We have implemented the technique and used it to establish the equivalence of some systems with state spaces too large for enumeration to be practical.

1. Introduction

Concurrent software is notoriously difficult to design and debug. Since such software is increasingly a part of safety critical systems, methods and tools for assuring its reliability are badly needed [12]. One of the most error-prone aspects of a concurrent system is the design of the communication protocol used by its cooperating agents. Fortunately, formal methods exist for aiding developers with the design and verification of communication protocols. Process algebras and labeled transition systems provide a foundation for representing and reasoning about concurrency. Analysis techniques based on these models, such as reachability analysis, are easily automated and can provide rigorous guarantees of a system's conformance to certain requirements. A reachability-based tool could, for example, prove a safety critical Ada tasking program free from communication deadlocks or other anomalies. In practice, however, the applicability of this kind of analysis is limited by the *state explosion problem*: the number of states that must be examined usually grows very quickly with the size of the program. Analysis of very large programs will undoubtedly require *compositional* techniques, which exploit the modularity of the program to reduce the complexity of the analysis.

Several techniques for compositional reachability analysis have already been proposed (e.g., [4, 5, 15]). The basic strategy of these techniques is to divide a large system into smaller subsystems, verify each subsystem, and then combine the results of these analyses to verify the full system. For concurrent processes, this is typically accomplished by decomposing the system into subsystems with simple interfaces, proving each subsystem is behaviorally equivalent to a simple process specifying the same interface with its environment, and then verifying the full system using the simpler interface processes. The decomposition is repeated until the subsystems are simple enough to analyze. For example, a system may contain a group of processes that implement a network protocol. Although these processes may have complex interactions with each other (e.g., because of sequence numbers, acknowledgements, etc.), their interface to the rest of the system (i.e., send packet, receive packet) is equivalent to a simple buffer.

The basis for a compositional analysis technique of this sort is a notion of equivalence between processes that allows a complex process, representing a subsystem, to be replaced by a simple process, representing the subsystem's interface with its environment. This equivalence is proven by either constructing a relation between the states of the subsystem process and the states of the interface process, or by hiding internal actions of the subsystem process and minimizing until it becomes the interface process. In any case, most existing algorithms for testing equivalences [6, 10] must enumerate the states of both processes.

In this paper, we consider the eventual application of compositional techniques to very large and complex programs. It seems likely that such programs will contain parts that cannot be further divided to yield a tractable analysis. Specifically, suppose the implementation I of a particular module of a program has n tasks: $I = I_1 \parallel \cdots \parallel I_n$. Further suppose that I contains far too many states to enumerate and that there is no subset of the I_i's that, when composed, has a simple interface with its environment (i.e., the tasks are highly coupled). The specification S of the module's interface with its environment is another process $S = S_1 \parallel \cdots \parallel S_m$ that may or may not be small enough to enumerate, but can be expressed as a composition of small processes. To use a compositional analysis, we must prove that S is equivalent to I, but we cannot use standard techniques since I (and perhaps S) have too many states to enumerate.

Here, we show that such an equivalence can be proven for a restricted class of systems. Specifically, we describe a technique for proving the trace equivalence of a specification process $S = S_1 \parallel \cdots \parallel S_m$ and an implementation process $I = I_1 \parallel \cdots \parallel I_n$ without enumerating the states of either S or I. The technique works from the component processes $S_1, \ldots, S_m, I_1, \ldots, I_n$ and involves the generation of necessary conditions for the existence of a trace showing that the equivalence does not hold. If these conditions cannot be satisfied, we may conclude that the equivalence holds. The necessary conditions are similar to those used in the constrained expression method for concurrent systems analysis [1, 2]. The use of necessary conditions results in a conservative analysis: the technique will never report that inequivalent processes are equivalent, but it may sometimes be unable to verify an equivalence that does hold.

Our technique is currently restricted to processes that are *deterministic* and *divergence-free*. Informally, a deterministic process has the property that the set of actions in which a process can engage is completely determined by the actions in which it has engaged previously. For example, a process representing a reliable network protocol should be deterministic: the packets it will deliver to the receiver should be completely determined by the packets it was given by the sender. A divergence-free process is one that cannot engage in an unbounded number of internal actions and thus ignore requests for interaction with its environment indefinitely. Most reactive systems are divergence-free. For nondeterministic processes, there is a hierarchy of many different kinds of equivalence (e.g., failure equivalence, observational equivalence), but for deterministic processes, this hierarchy collapses and all forms of equivalence are the same as trace equivalence.

We have implemented our technique and used it to establish the trace equivalence of some systems with state spaces too large for enumeration to be practical. The only other non-enumerative equivalence techniques we know of use Ordered Binary Decision Diagrams (OBDDs) to represent the states. Examples of this work are [3] and [9]. The successful application of OBDD-based methods depends heavily on finding a good ordering for the state variables. While some heuristics exist, especially for models of certain digital circuits, the successful application of these methods to concurrent software systems with highly coupled tasks remains extremely problematic.

This paper is organized as follows. Section 2 presents basic definitions used throughout. Section 3 describes the method and Section 4 presents the results of some experiments with the method. Finally, Section 5 concludes and discusses future directions for this work.

2. Definitions

Formally, we regard a process as a finite-state automaton in a standard fashion. States of a process can be regarded as tuples giving the values of all relevant local variables, including the program counter, and a transition between states corresponds to an event changing one or more of these variables.

Definition 1. A *process* P is a quadruple (S, Σ, Δ, s_P) where S is a set of states, Σ is an alphabet, $\Delta \subseteq S \times \Sigma \times S$ is a transition relation, and $s_P \in S$ is the start state.

A concurrent system is usually thought of as a collection of tasks or threads of control, each of which might be modeled by an individual process. We use the term "process" in the sense of process algebra, so our processes might represent single tasks/threads or a group of tasks/threads executing concurrently. We define a parallel composition operator that constructs a single process, representing a concurrent system, from the processes representing the tasks that comprise it. When processes are composed in parallel, the resulting process may accept a symbol if and only if every component containing that symbol can accept the symbol.

Definition 2. Given $P_1, \ldots, P_n$ where $P_i = (S_i, \Sigma_i, \Delta_i, s_{P_i})$, the parallel composition $P_1 \parallel \cdots \parallel P_n$ is defined as another process $P = (S, \Sigma, \Delta, s_P)$ where: $S = S_1 \times \cdots \times S_n$, $\Sigma = \bigcup_i \Sigma_i$, $s_P = (s_{P_1}, \ldots, s_{P_n})$, and $((s_1, \ldots, s_n), a, (s'_1, \ldots, s'_n)) \in \Delta$ iff for all $i = 1, \ldots, n$:

$$(a \notin \Sigma_i \wedge s_i = s'_i) \vee (a \in \Sigma_i \wedge (s_i, a, s'_i) \in \Delta_i)$$

We assume processes communicate in pairs (as in Ada and CSP). Given a subsystem represented as a parallel composition $P = P_1 \parallel \cdots \parallel P_n$, this divides the symbols in Σ into three disjoint sets:

Internal Actions: (Σ_{int}) A symbol representing an internal action of a single P_i is in the alphabet of only that process.

Communication Actions: (Σ_{com}) A symbol representing communication between two P_i is in the alphabets of exactly those two processes, one of which is denoted the *caller*, and the other the *acceptor*.

Visible Actions: (Σ_{vis}) A symbol representing an unmatched communication with the environment is in the alphabet of exactly one P_i. Although they represent communications, we do not include visible actions in Σ_{com}.

When processes are composed, some of their actions cease to be interesting from an external point of view. In most process algebras, such actions are hidden by a special operator that renames them to a special "invisible" action τ. Here, we take the approach of Valmari [14] and specify a set $\Sigma_{vis} \subseteq \Sigma$ of visible actions (unmatched communications). Only the visible actions of a process are considered when proving equivalence. With most techniques for proving equivalence, a process $P = P_1 \parallel \cdots \parallel P_n$ is considered a single composed entity, thus the actions in Σ_{com} and Σ_{int} are not distinguished since they all represent actions internal to P. In our technique, we do not construct P but work from its components $P_1 \parallel \cdots \parallel P_n$, thus we require the distinction between Σ_{com} and Σ_{int}. The following definitions are adapted from [14].

Definition 3. Let $P = (S, \Sigma, \Delta, s_P)$ be a process and $\Sigma_{vis} \subseteq \Sigma$ be a set of visible actions.

- For $\sigma \in \Sigma^*$, $vis(\sigma)$ is the projection of σ onto Σ_{vis} (i.e., all invisible actions are removed).
- $s \xrightarrow{\sigma} s'$, where $\sigma = a_1 \ldots a_m \in \Sigma^*$, iff $\exists s_0, \ldots, s_m \in S$ such that $s_0 = s$, $s_m = s'$, and $(s_{i-1}, a_i, s_i) \in \Delta$ for $i = 1, \ldots, m$.
- $s \xrightarrow{\sigma}$, where $\sigma = a_1 \cdots \in \Sigma^\omega$, iff $\exists s_0, \ldots \in S$ such that $s_0 = s$ and $(s_{i-1}, a_i, s_i) \in \Delta$ for $i = 1, \ldots$
- A process P is *divergence-free* iff $\neg\exists \sigma \in \Sigma^*, \rho \in (\Sigma - \Sigma_{vis})^\omega$ such that $s_P \xrightarrow{\sigma} s, s \xrightarrow{\rho}$ for some $s \in S$.
- For $s \in S$, $next(s) = \{a \in \Sigma | \exists s' \in S : \ s \xrightarrow{a} s'\}$.
- $s \overset{\sigma}{\Rightarrow} s'$ iff $\exists \sigma' \in \Sigma^*$ such that $s \xrightarrow{\sigma'} s'$ and $\sigma = vis(\sigma')$.
- A process P is *deterministic* iff, for all $s \in S$ and $\sigma \in \Sigma^*$, $s \xrightarrow{\sigma} s'$ and $s \xrightarrow{\sigma} s''$ implies $next(s') = next(s'')$.
- $Traces(P) = \{\sigma \in \Sigma^*_{vis} | \exists s \in S : \ s_P \overset{\sigma}{\Rightarrow} s\}$.
- Processes P and P' are *trace equivalent* iff $Traces(P) = Traces(P')$.

3. Method

Most techniques for proving the equivalence of two processes involve refining a partition of their state sets until all the states in each part are equivalent. Our approach is to generate necessary conditions for the existence of a counterexample showing that the equivalence does not hold. To prove that two processes are not trace equivalent, we generate necessary conditions for the existence of a trace of one that is not a trace of the other. If these conditions cannot be satisfied, we may conclude that the equivalence holds. If the conditions can be satisfied, however, the two processes may still be trace equivalent, since the conditions are necessary but not sufficient. To be useful, the conditions must be strong enough so that they are usually not satisfiable if the equivalence holds. Furthermore, generating the conditions and checking their satisfiability must be tractable.

If two processes are not trace equivalent, there must be a trace of one that is not a trace of the other. Given such a ρ of length at least 1, let σ be the longest prefix of ρ that is a trace of both processes. Then $\rho = \sigma a_1 a_2 \dots a_k$, with $k > 0$, and σa_1 is a trace of one system but not of the other. This motivates the following definition.

Definition 4. Let $\sigma \in \Sigma_{vis}^*$ and $a \in \Sigma_{vis}$. We say that the pair (σ, a) is *bad* for processes S and I if $\sigma \in \mathit{Traces}(S) \cap \mathit{Traces}(I)$ and

(i) $\sigma a \notin \mathit{Traces}(I)$ and $\sigma a \in \mathit{Traces}(S)$, or
(ii) $\sigma a \notin \mathit{Traces}(S)$ and $\sigma a \in \mathit{Traces}(I)$.

Processes S and I are not trace equivalent if and only if there exists a pair (σ, a) that is bad for S and I.

We now describe the generation of necessary conditions for the existence of a bad pair for processes S and I. If these conditions are inconsistent, we know that no bad pair can exist and therefore that S and I are trace equivalent. Generating these necessary conditions does not require the construction of $S = S_1 \parallel \cdots \parallel S_m$ or $I = I_1 \parallel \cdots \parallel I_n$, but works from the component processes $S_1, \dots, S_m$ and $I_1, \dots, I_n$. Any mention of states and transitions in the following description refers to the states and transitions of the component processes $S_1, \dots, S_m$ and $I_1, \dots, I_n$.

To generate the necessary conditions, we adapt the basic integer programming technique described in [1, 2]. Given a set of communicating processes, that technique uses necessary conditions, in the form of linear inequalities, to either help find a trace with certain properties or prove that no such trace could exist. A trace can be viewed as a path in each component process beginning at the start state such that the interactions between the processes represented by the paths are consistent. Our technique finds a flow in each process such that the flows satisfy a weaker consistency criterion. Specifically, we require that, for each communication symbol, the processes containing that symbol agree on the number of times that they accepted the symbol.

The basic technique produces a system of inequalities representing conditions that must be satisfied by any trace. First, we assign a *transition variable*, x_i, to each transition i that represents the number of times transition i is taken in the trace. We also assign a *connection variable*, c_j, to each state j that will be one if the process is in that state at the end of the trace, and zero otherwise. We then generate a *flow equation* for each state, equating the flow into the state with the flow out of the state (i.e., the number of times the state is entered equals the number of times it is exited). There is an implicit flow in of one at the start state and the connect variables are counted as flow out. Finally, we generate a *communication equation* for each communication symbol, equating the number of times the tasks containing the symbol accepted it. It is easy to show that the resulting inequality system has an integral solution for every trace.

Using this basic technique, we generate the necessary conditions for the existence of a bad pair as follows. We generate separate systems of inequalities representing necessary conditions for *(i)* and *(ii)* of Definition 4; since these are essentially the same except for exchanging S and I, we describe here only the method for *(i)*. To generate necessary conditions for the existence of $\sigma \in \Sigma_{vis}^*, a \in \Sigma_{vis}$ such that $\sigma a \in Traces(S)$ and $\sigma a \notin Traces(I)$, we generate the following inequalities:

- $\exists \sigma \in Traces(S)$ and $\exists \sigma' \in Traces(I)$. We use the basic technique to generate flow and communication inequalities representing necessary conditions for the existence of traces of S and I.
- $\sigma = \sigma'$. We generate a *prefix consistency equation* for each $a \in \Sigma_{vis}$ equating the number of occurrences of a in σ with the number of occurrences of a in σ'. Like the communication equations, this enforces a weak consistency between σ and σ' and represents a necessary condition for $\sigma = \sigma'$.
- $\exists a \in \Sigma_{vis}$. To select the extension $a \in \Sigma_{vis}$, we create an *extension variable*, e_a, for each $a \in \Sigma_{vis}$, that will be one if a is the extension. We then generate an *extension selection equation* summing the extension variables to one.
- $\sigma a \in Traces(S)$. We generate an *extension enabled inequality* for each $a \in \Sigma_{vis}$ that allows an extension symbol to be selected after σ only if some component of S is in a state with a transition on that symbol.
- $\sigma' a \notin Traces(I)$. We generate an *extension excluded inequality* for each state in a component of I that allows the component to be in that state after σ' only if there are no transitions on a. However, we must exclude the possiblity of stopping in a state with no a transitions when a state having a transitions can be reached through a sequence of invisible actions. To achieve this, we generate *prefix progress inequalities* to prevent the components of I from stopping in a state where some invisible action is enabled. For each state j of a component of I with an internal action $a \in \Sigma_{int}$ possible, we set $c_j = 0$, thereby preventing the flow from stopping in state j. For each communication action $b \in \Sigma_{com}$, we generate an inequality that prevents the caller and acceptor of b from both stopping in states where b is possible.

The algorithm for generating the necessary conditions for the existence of a bad pair is shown in Figure 1. Table 1 gives a list of abbreviations used in the algorithm. Figure 2 gives a small example of a two-slot buffer implemented as the parallel composition of two one-slot buffers and the inequalities generated by the algorithm in Figure 1. For this example, $\Sigma_{vis} = \{a, b\}$ where a represents data being put into the buffer, b represents data being removed, and $c \in \Sigma_{com}$ represents the data being passed between the two one-slot buffers. The inequalities have no integral solution, proving that no bad pair satisfying condition *(i)* exists for this example.

The prefix progress inequalities require some further comment. We are generating necessary conditions for $\sigma' a$ to *not* be a trace of I, while σ' is a trace of I. Our conditions require flows through the components of I that are compatible with σ', in the sense of having each action occur the right number of times, and that reach a state where it is not possible for an a action to occur. Even if an a action cannot occur in the state reached after σ', it may be possible to execute a after a sequence of invisible actions. Since we want to prohibit a from being the next *visible* action,

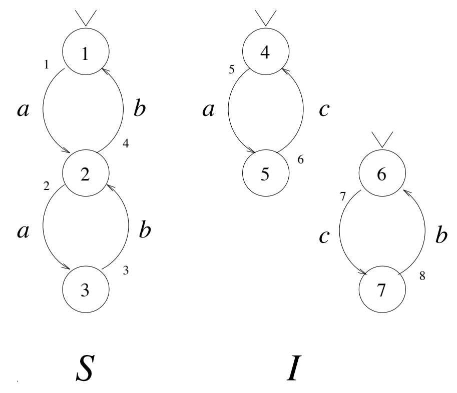
```
Input:   Processes $S_1, \dots, S_m$ comprising $S$
         Processes $I_1, \dots, I_n$ comprising $I$
         $\Sigma_{vis}$ = visible actions of $S$ and $I$
         $\Sigma_{com}$ = communication actions between $S_1, \dots, S_m$ or $I_1, \dots, I_n$
         $\Sigma_{int}$ = internal actions of $S_1, \dots, S_m, I_1, \dots, I_n$
Output:  A set of inequalities representing necessary conditions for the existence
         of a pair $(\sigma, a)$ with $\sigma a \notin \mathit{Traces}(I)$ and $\sigma a \in \mathit{Traces}(S)$

For each transition $k$ of a process:
    Create transition variable $x_k$
For each state $j$ of a process:
    Create connection variable $c_j$
For each symbol $a \in \Sigma_{vis}$:
    Create extension variable $e_a$

For each state $j$ of a process:
    Generate flow equation: $\mathit{start(j)} + \sum_{k \in in(j)} x_k = \sum_{k \in out(j)} x_k + c_j$
For each symbol $a \in \Sigma_{com}$:
    Generate communication equation: $\sum_{k \in lab(a,call(a))} x_k = \sum_{k \in lab(a,acc(a))} x_k$

    Generate prefix progress inequality: $\sum_{j \in enab(a,call(a))} c_j + \sum_{j \in enab(a,acc(a))} c_j \leq 1$

Generate extension selection equation: $\sum_{a \in \Sigma_{vis}} e_a = 1$
For each symbol $a \in \Sigma_{vis}$:
    Generate prefix consistency equation: $\sum_{k \in lab(a,S)} x_k = \sum_{k \in lab(a,I)} x_k$

    Generate extension enabled inequality: $e_a \leq \sum_{j \in enab(a,S)} c_j$
For each state $j$ of a process of $I$:
    If $\exists a \in \mathit{next}(j) \cap \Sigma_{int}$ then
        Generate prefix progress equation: $c_j = 0$
    Else
        Generate extension exclusion inequality: $c_j + \sum_{a \in next(j)} e_a \leq 1$
```

FIGURE 1. Algorithm for generating inequalities

we use the prefix progress inequalities to eliminate any flows leading to a state of I in which some invisible action is possible. If an infinite sequence of invisible actions is possible after σ', these inequalities may eliminate all possible flows compatible with σ' even if (σ', a) is a bad pair. Thus if I is not divergence-free, our inequalities are not necessary conditions for the existence of a bad pair. Our method only

acc(a)	process that is acceptor for communication symbol a
call(a)	process that is caller for communication symbol a
next(j)	set of event symbols labeling transitions out of state j
enab(a,P)	set of states of process P having a transition labeled a
in(j)	set of transitions into state j
lab(a,P)	tranitions of process P labeled with event a
out(j)	set of transitions out of state j
start(j)	1 if state j is a start state of a process, else 0

TABLE 1. Abbreviations Used in Algorithm



Flow: **(state)**

(1) $1 + x_4 = x_1 + c_1$

(2) $x_1 + x_3 = x_2 + x_4 + c_2$

(3) $x_2 = x_3 + c_3$

(4) $1 + x_6 = x_5 + c_4$

(5) $x_5 = x_6 + c_5$

(6) $1 + x_8 = x_7 + c_6$

(7) $x_7 = x_8 + c_7$

Communication: **(symbol)**

(c) $x_6 = x_7$

Prefix Progress: **(symbol)**

(c) $c_5 + c_6 \leq 1$

Extension Selection:

$e_a + e_b = 1$

Prefix Consistency: **(symbol)**

(a) $x_1 + x_2 = x_5$

(b) $x_3 + x_4 = x_8$

Extension Enabled: **(symbol)**

(a) $e_a \leq c_1 + c_2$

(b) $e_b \leq c_2 + c_3$

Extension Exclusion: **(state)**

(4) $c_4 + e_a \leq 1$

(5) $c_5 \leq 1$

(6) $c_6 \leq 1$

(7) $c_6 + e_b \leq 1$

FIGURE 2. Two Slot Buffer Example

applies, then, in the case in which both S and I are divergence-free (note that the component processes $I_1, \ldots, I_n$ and $S_1, \ldots, S_m$ need not be divergence-free). The techniques of [7] can be used to check for this without enumerating the states of S and I.

Theorem 1. *If S and I are divergence-free and the inequality system generated by the algorithm in Figure* 1 *has no integral solutions, no bad pair exists for S and I. It follows that S and I are trace equivalent.*

Proof. If there exists a bad pair (σ, a), then we may construct an integral solution to the inequality system as follows. We set e_a to one and all other extension variables to zero. Since $\sigma \in Traces(S) \cap Traces(I)$, there exists a sequence of transitions in each component of S and I that defines a path through that component. We set each x_k to the number of times transition k is taken in these paths, and set c_j to the one if the process in in state j at the end of its path, otherwise we set it to zero. It is easy to check that this assignment will satisfy all the inequalities generated. □

Although our inequalities represent necessary conditions for the existence of a bad pair, they are not sufficient conditions for several reasons. First, the communication equations do not guarantee that there is a consistent ordering of the communication symbols within the parallel compositions of S and I (e.g., one process could accept a then b, while the other accepted b then a). Similarly, the prefix consistency equations do not guarantee that there is a consistent ordering of the visible actions between the trace found for S and the trace found for I. Thirdly, the presence of cycles in the processes can allow cyclic flows that are not connected to the path found within the process (e.g., if we added transition 9 from state 3 to itself in the S process of Figure 2, then the flow equation for state 3, $x_9+x_2 = x_3+x_9+c_3$, allows $x_9 = 1$ even if state 3 is never visited).

Solutions to our inequalities may also arise due to nondeterminacy of S or I even when no bad pair exists. This can occur because, in a nondeterministic system, it may be possible to reach two states after a sequence of actions σ', with a possible in one state and not possible in the other. Thus there would exist a solution to our inequalities corresponding to flows leading to a state in which a is not possible after σ' even though $\sigma' a$ is a trace. Although this problem will not lead to incorrect reports of equivalence, it means that our analysis is unlikely to be useful with nondeterministic processes.

For these reasons, a solution to the inequality system may not correspond to a bad pair. If such a solution is found, the analysis is inconclusive since the presence of that solution implies nothing about the existence of another solution that does correspond to a bad pair. In our experience with these conditions, we have found that they are strong enough to verify equivalences of interesting systems, although a much more extensive empirical investigation would be necessary to characterize their applicability in general. In addition, we note that the technique of [7] can be used to eliminate some spurious solutions. That technique involves splitting the trace into segments, using the basic technique to generate an inequality system for each segment, and then connecting the inequality systems together to form necessary conditions for the entire trace. By placing restrictions on the segments, we may prevent certain spurious matchings of communication actions or disconnected cyclic flows at the price of larger inequality systems. For example, the method presented here was unable to prove the equivalence of the stop and wait protocol to a simple buffer. A spurious solution resulted from the loss of information on the order of certain events. By splitting the trace into two segments, however, we were able to strengthen the conditions and prove the equivalence. We are currently exploring the range of applicability of this technique.

The worst case complexity of our algorithm is no better than that of those methods employing state enumeration. The complexity of checking trace equivalence for deterministic processes is polynomial in the number of states in the processes. Unfortunately, the number of states in a process $P = P_1 \parallel \cdots \parallel P_n$ is often proportional to the *product* of the sizes of the components, and is thus exponential in n. Our technique produces an inequality system whose size is proportional to the *sum* of the sizes of the components. In particular, the number of variables/inequalities is proportional to the number of transitions/states in the components. We then, however, apply integer linear programming (ILP), which is an *NP*-hard problem for which we use an exponential-time decision procedure. Despite the complexity of ILP, our experience with these kinds of inequality systems [2] suggests that they are easier to solve than the general case, probably because a large part of the systems are network flow equations (pure network flow systems can be solved in polynomial time).

We conclude this section by noting that, when the equivalence does not hold, the solution to the inequality system is usually helpful in showing why. For all of the non-equivalent processes to which we have applied the method, the solution obtained provided an example of a bad pair.

4. Experiments

We have demonstrated the feasibility of our method by implementing it and applying it to several examples. Here, we describe experiments on two scalable examples. To conduct these experiments, we modified the Inequality Necessary Condition Analyzer (INCA), an analysis tool for concurrent and real-time systems that is descended from the constrained expression toolset [2]. INCA takes as input a set of specification and implementation tasks specified in an Ada-like design language. The tool translates the tasks into FSAs and then produces an inequality system using the algorithm in Figure 1. The inequality system is solved using the IMINOS optimization package and INCA then interprets the solution, if any. All times we report are in CPU seconds on a SPARCstation 10 Model 41 with 64MB of memory and include both user and system time.

The first example is a scaled version of the compositional buffer example shown in Figure 2. Table 2 shows the results of proving that n one-slot buffers composed end-to-end in parallel are trace equivalent to a single n-slot buffer. The columns of the table show the problem size (n), the number of inequalities and variables generated, the time to generate the inequality system, the time to solve the system, and the total time. The two rows for each size represent the analyses for conditions *(i)* and *(ii)* of Definition 4, respectively.

The second example is adapted from a real world problem reported in [11] and recently studied in the concurrency and distribution track of the Sixth International Workshop on Software Specification and Design. This router problem models a communication network with M input ports, each connected to a sender, and N output ports, each connected to a receiver. Each sender repeatedly picks some receiver and sends a message to that receiver through the network. The network is implemented as an M by N grid of switching elements as shown in Figure 3. Messages travel across the row of their sender and then down the column of their receiver. In the original problem, messages were divided into multiple packets and no flow control was specified. For our experiments, we used only single-packet messages and assumed that a sender would wait for an acknowledgement (which retraced the path of the message through the network) before sending another message. Only the latter simplification was essential. Without it, the network could buffer up to $M \times N$ messages and would not have a simple specification.

The router problem is interesting in that the external behavior of the system is very simple but it is difficult to decompose the system into subsystems with a simple visible behavior. We may specify the externally visible behavior of the router

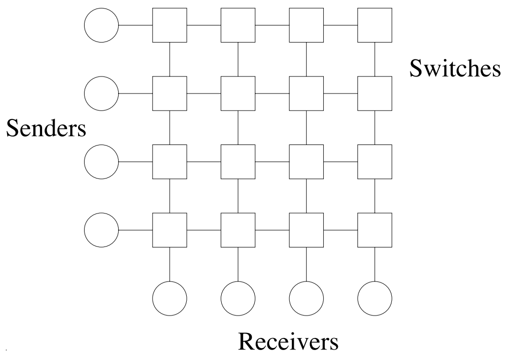


FIGURE 3. Router Implementation

	Size		Time		
Size	Ineqs	Vars	Gen	Solve	Total
100	407	411	34	3	
	407	411	34	3	74
200	807	811	86	6	
	807	811	87	7	186
300	1207	1211	166	9	
	1207	1211	166	12	353
400	1607	1611	273	14	
	1607	1611	275	19	581
500	2007	2011	412	19	
	2007	2011	417	29	877

TABLE 2. Performance on Compositional Buffer

	Size		Time		
Size	Ineqs	Vars	Gen	Solve	Total
2	49	38	6	1	
	49	38	6	1	14
4	185	156	32	3	
	185	156	33	3	71
6	409	354	140	12	
	409	354	142	13	307
8	721	632	525	36	
	721	632	529	37	1127
10	1121	990	1599	84	
	1121	990	1621	81	3385

TABLE 3. Performance on Router

as the parallel composition of M processes, $S_1, \ldots, S_M$, each representing one of the senders. Each of these specification processes accepts a visible action from the environment requesting delivery of a message to a specific output port, then accepts a visible action representing the delivery of the message by the intended receiver. Although the number of states in the specification process S is N^M, the size of each S_i is $O(N)$.

The results of applying our method to prove a specification of the router problem trace equivalent to an implementation are shown in Table 3. We used a square grid so $n = M = N$. The number of states in the specification of the size n version of the problem is at least n^n, and the number of states in the implementation is much greater. The size of the inequality system is growing quadratically in n (linearly in the number of components) and analysis times seem to be growing somewhat faster, but much slower than n^n. The larger sizes we ran are well out of the range of state enumeration-based techniques for proving equivalence, which typically can handle at most between 10^5 and 10^6 states. We do not know whether a symbolic

OBDD-based technique would be capable of proving the equivalence; however, we believe that the two-dimensional nature of the problem would make it difficult to find a good variable ordering for the OBDDs.

5. Conclusion

We have taken a step towards applying compositional analysis to very large systems by providing a method for proving the equivalence of processes without enumerating their states. The strength of the method lies in its potential applicability to very large processes for which existing enumeration-based approaches would be intractible. The weakness of the method is that it is not always applicable; currently, it works only for deterministic divergence-free processes and the necessary conditions it employs may be too weak to prove the equivalence even if it holds. Nevertheless, we have successfully applied the method to examples that are far too large for existing enumeration-based approaches.

We are currently exploring ways to strengthen the conditions and extend their applicability. As noted in Section 3, we are investigating the applicability of a technique for strengthening the conditions by splitting the trace into multiple segments. We are also exploring ways to extend the method to nondeterministic processes and stronger kinds of equivalence (e.g., failure equivalence) by using different flow equations that explore all possible paths.

Acknowledgements

This research was partially supported by grants from the National Science Foundation and the Office of Naval Research.

References

1. G. S. Avrunin, U. A. Buy, and J. C. Corbett. Integer programming in the analysis of concurrent systems. In K. G. Larsen and A. Skou, editors, *Computer Aided Verification, 3rd International Workshop Proceedings*, volume 575 of *Lecture Notes in Computer Science*, pages 92–102, Aalborg, Denmark, July 1991. Springer-Verlag.
2. G. S. Avrunin, U. A. Buy, J. C. Corbett, L. K. Dillon, and J. C. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Trans. Softw. Eng.*, 17(11):1204–1222, Nov. 1991.
3. A. Bouali and R. de Simone. Symbolic bisimulation minimisation. In v. Bochmann and Probst [13], pages 97–108.
4. S. C. Cheung and J. Kramer. Enhancing compositional reachability analysis using context constraints. In *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 115–125, Dec. 1993.
5. E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, 1989.
6. R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics based tool for the verification of concurrent systems. *ACM Trans. Prog. Lang. Syst.*, 15(1):36–72, Jan. 1993.
7. J. C. Corbett. Verifying general safety and liveness properties with integer programming. In v. Bochmann and Probst [13], pages 357–369.
8. C. Courcoubetis, editor. *Computer Aided Verification, 5th International Conference Proceedings*, Elounda, Greece, 1993.
9. J. C. Fernandez, A. Kerbrat, and L. Mounier. Symbolic equivalence checking. In Courcoubetis [8], pages 85–96.
10. P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86:43–68, 1990.

11. D. May and P. Thompson. Transputers and routers: Components for concurrent machines. In T. L. Kunii and D. May, editors, *Proceedings of the Third Transputer/Occam International Conference*, Tokyo, May 1990. IOS Press.
12. L. Osterweil and L. Clarke. A proposed testing and analysis research initiative. *IEEE Software*, pages 89–96, Sept. 1992.
13. G. v. Bochmann and D. K. Probst, editors. *Computer Aided Verification, 4th International Workshop Proceedings*, volume 663 of *Lecture Notes in Computer Science*, Montreal, Canada, 1992. Springer-Verlag.
14. A. Valmari. On-the-fly verification with stubborn sets. In Courcoubetis [8], pages 397–408.
15. W. J. Yeh and M. Young. Compositional reachability analysis using process algebra. In *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4)*, pages 178–187, New York, Oct. 1991. ACM SIGSOFT, Association for Computing Machinery.

Information and Computer Science Department, University of Hawaii at Manoa, Honolulu, HI 96822
E-mail address: corbett@hawaii.edu

Department of Mathematics, Box 34515, University of Massachusetts at Amherst, Amherst, MA 01003-4515
E-mail address: avrunin@math.umass.edu