Polytechnic
UNIVERSITY
Brooklyn · Long Island · Westchester

Extending FLAVERS to Check Properties on Infinite Executions of Concurrent Software Systems

Gleb Naumovich Lori A. Clarke

Department of Computer and Information Science

Technical Report
TR-CIS-2000-02
04/14/2000

Extending FLAVERS to Check Properties on Infinite Executions of Concurrent Software Systems

Gleb Naumovich
Polytechnic University, Brooklyn
Department of Computer and Information Science
Brooklyn, NY 11201
(718) 260-3554
gleb@poly.edu

Lori A. Clarke
Computer Science Department
University of Massachusetts
Amherst, Massachusetts 01003
(413) 545-2013
clarke@cs.umass.edu

1. Introduction

Finite state verification techniques can be used for detecting the presence or proving the absence of certain kinds of errors in software systems. These approaches are based on reasoning about a finite, abstracted model of a system's behaviors. FLAVERS (FLow Analysis for VERification of Systems) is a finite state verification approach that uses data flow analysis techniques to verify user-specified properties of sequential and concurrent software systems [11, 12]. FLAVERS is capable of verifying properties about sequences of events, where the events are recognizable actions in the program and where the sequences are either translated into or specified directly as a finite state automaton (FSA). The attractiveness of this approach is in its low-order polynomial complexity bounds, its ability to derive the model of executable behavior automatically from a program's source code, and its ability to improve the precision of the analysis by incrementally improving the accuracy of the program model. For concurrent programming languages, FLAVERS prototypes have been developed for Ada [11] and Java [16].

To date, FLAVERS has been restricted to considering programs with only finite executions. This is a serious limitation, because, in practice, distributed systems are frequently intended to execute infinitely. In this paper we propose two extensions to the FLAVERS analysis algorithm that allows FLAVERS to check properties on software systems with infinite executions.

Traditionally, verification properties have been classified into two broad categories: safety and liveness [1, 3]. The distinction is that safety properties are finitely refutable and liveness properties are never finitely refutable. Intuitively, a safety property specifies that an undesirable state of the system is never reached and a liveness property specifies that a desirable state of the system is eventually reached on all executions.

Any property can be represented as a union of a safety property and a liveness property [2]. Any property checked only on finite executions of a system is a safety property, since the so-called "undesirable" state of the system can be viewed as the terminal state[1] where the predicate of interest either holds or doesn't hold. Safety properties are also a concern for infinite executions, when the property is finitely refutable. Thus, a property that can be proved by examining all finite execution trace prefixes would be such an infinite safety property. In this paper we focus on extending FLAVERS to handle *both safety and liveness* properties for *infinite* executions.

Our extension for checking safety properties on infinite executions is very simple and requires two small modifications to the original FLAVERS' approach: (1) a modification of the property representation and (2) a simple change in the analysis algorithm, where property violations are checked not only in the terminal state of the system but also at relevant intermediate points of system execution.

Our extension for checking liveness properties with FLAVERS is based on representing the property of interest as a Büchi automaton [22] and computing the states that this automaton can be in at different points of the program execution. This computation is carried

[1] Without loss of generality, we can assume that there is a single terminal state.

out by the same data flow algorithm that FLAVERS uses for checking properties on finite executions. After that we determine if the graph contains infinite paths with suffixes on which the property Büchi automaton never enters an accept state. An existence of such a path signifies that the property represented by the automaton does not hold on the execution of the system corresponding to this path through the graph. Checking this is based on computing maximal strongly-connected components in the derived representation of the program. This algorithm could also be applied to safety properties for infinite executions, but its worst-case bound is larger than that of the algorithm specialized for safety properties on infinite executions.

Our algorithm for checking safety properties on infinite executions uses a form of FSA-based property specification, used, for example, in [7] and makes a fairly obvious change to the original data flow algorithm of FLAVERS. Our algorithm for checking liveness properties is similar to the existing algorithms used by model checking [9] and reachability analysis [5,13] approaches. Despite these similarities, we make several important contributions. First, our proposed extensions maintain the current strong points of FLAVERS, using an efficient data flow algorithm, automatically dealing with software systems at the implementation level (although FLAVERS can handle high-level specifications as well [17]), and giving the analyst the opportunity to improve the precision of the analysis incrementally by deferring the modeling of certain features of the system until it becomes clear that such modeling is necessary. Second, we can use the existing FLAVERS framework for specifying the appropriate conditions that should be assumed during infinite executions of the system. Finally, we do not assume that all loops in the threads of control of the system can execute infinitely. Instead, the analyst has the means of specifying which of the thread loops can or cannot execute infinitely.

For convenience, we introduce the following abbreviations. We will refer to the original algorithm of FLAVERS [11] as *finite executions*, or *FE*, algorithm; to the proposed algorithm for checking safety properties on infinite executions as *safety infinite executions*, or *SIE*, algorithm; and to the proposed algorithm for checking liveness properties as *liveness infinite executions*, or *LIE*, algorithm.

In the next section we give a brief overview of the existing techniques for checking liveness properties on infinite executions of software systems. In Section 3 we describe the *FE* algorithm. Section 4 describes the specification of safety properties on infinite executions and introduces the *SIE* algorithm. In Section 5 we describe the specification of liveness properties and introduce the *LIE* algorithm. In Section 6 we discuss some the issues related to fairness conditions and incremental precision improvements. Finally, in Section 7, we outline directions for future work.

2. Related work

There exists a considerable amount of work on finite state verification approaches for verifying concurrent systems. There are four major approaches to finite state verification: reachability analysis, necessary conditions, model checking, and data flow analysis approaches. In this section we describe the way in which some representative finite state verification techniques handle properties on infinite executions.

SPIN [13] is a reachability analysis technique that accepts properties expressed in linear temporal logic (LTL) and focuses on systems with asynchronous concurrency control. Each of the threads of control in the software system is modeled with a Büchi automaton and the negation of a property is also represented as a Büchi automaton. All these Büchi automata are combined in a synchronous cross-product, with the worst-case size of this automaton being exponential in the number of threads of control. If the language of the resulting Büchi automaton is non-empty, it means that the property can be violated. This can be determined in time linear in the number of states and transitions in the combined Büchi automaton by performing the Tarjan depth-first search algorithm [20] for constructing all strongly-connected components. If there exists a reachable strongly-connected component that contains at least one accepting state, a reachable acceptance cycle exists, and so the property violation is found. The complexity of SPIN analysis is $\mathcal{O}(S+V)$, where S is the number of states in the product Büchi automaton and V is the number of transitions in this automaton.

Enhanced Compositional Reachability Analysis (ECRA) [8] works similar to SPIN. ECRA computes a cross product of Büchi automata in a compositional way, "hiding" some of the transitions and thus, potentially, reducing the size of the cross product automaton. For safety properties, ECRA uses FSAs to represent all threads of control as well as to represent the property, which is augmented with a special trap state that represents property violations. After the FSAs for the property and threads of control are composed into a single cross-product automaton, the property is violated if this cross-product automaton contains a trap state. The analysis of liveness properties with ECRA is done in a similar way [5], using Büchi automata instead of FSAs. The property is considered to be violated if there is a reachable strongly-connected component in the product Büchi automaton that does not contain transitions to accepting states of this automaton.

Conversely, if each reachable strongly-connected component contains at least one transition into an accepting state of the product automaton, the property holds. The latter holds true only under a relatively strong fairness assumption that each transition in a reachable strongly-connected component is eventually executed if this strongly-connected component is executed forever (the semantics of the distributed systems for which ECRA is designed make this assumption possible). The worst-case complexity of ECRA is the same as that of SPIN, but a good decomposition of the system model in practice may result in significant reductions in the size of the product automaton.

Necessary conditions analysis [10] generates a set of integer linear inequalities that represents necessary conditions for the existence of infinite system executions for systems with synchronous concurrency control. The necessary conditions express constraints on the number of times certain system events take place relative to other system events. Some of the constraints are computed using strongly-connected components of Büchi automata, where each automaton represents a thread of control in the system. In case of liveness properties, some additional inequalities have to be introduced in the system. The negation of the property is also represented as a set of inequalities. Integer linear programming is used to solve this system of inequalities. If no solution of the system of the resulting inequalities exists, the property holds on all executions of the system. This approach is NP-hard in the size of the system of integer inequalities that has to be solved. In practice, this approach is often very efficient, although it does not appear to be applicable to asynchronous communication mechanisms.

Model checking [9] does not make a clear distinction between safety and liveness properties. In this approach, it is assumed that all executions of the system are infinite and properties are represented in computation tree logic (CTL). To prove a CTL formula F, model checking constructs a Kripke structure [14] for the system. This Kripke structure represents the set of all reachable states of the system and thus the number of its states is exponential in the number of threads of control and modeled variables. The goal of model checking is to check whether or not formula F holds in the start state of the Kripke structure, which signifies the start of all possible executions of the system.

In general, the complexity of model checking is $\mathcal{O}(f(V+E))$, where f is the size of the CTL formula representing the property. The algorithm for checking liveness properties with FLAVERS that we propose in this paper is quite similar to this specific case of model checking.

3. The *FE* version of FLAVERS

In this section we introduce the FSA-based property specification used by FLAVERS, give a very high-level overview of FLAVERS, and then present the *FE* algorithm.

3.1. Representing *FE* Properties

FLAVERS uses an event-based view of the software system being analyzed. In this view, user-selected names, called *events*, are associated with observable activities of interest in the system and then all potential executions of the system are represented as sequences of these events. For example, both a variable assignment and a method call could be examples of events.

A number of formalisms for specifying properties have been proposed, including temporal logics [9, 19], process algebras [4, 15], and various forms of regular languages and finite state automata [18, 21]. FLAVERS uses deterministic finite state automata for specifying properties to be checked on terminating executions of a system.

A deterministic FSA can be represented as a tuple $(S, s_0,$
$\Sigma, \delta, A)$. S is the set of all *states* of the FSA, including the unique start state s_0. Σ is called the *alphabet* of the FSA and includes all events used by this FSA. δ is a total *transition function* $S \times \Sigma \rightarrow S$ that represents all event-based transitions between the states of the FSA. We deal with *total* FSAs, which means that from any state there is a transition based on each event from the alphabet. We write $\delta(s, e) = s'$ to indicate that there is a transition from state s to state s' based on event e. Finally, A is the set of *accept* states $\{a_1, a_2, ..., a_p\}$, $\forall 1 \leq i \leq p, a_i \in S$. A *trace* of an FSA on an event sequence $w = e_1, e_2, ..., e_n$ is a sequence of states $s_0, s_1, ..., s_n$, where s_0 is the start state and for any $i, 1 \leq i \leq n$ there is a transition from s_{i-1} to s_i on event e_i. A sequence of events $e_1, ..., e_n$ is *accepted* by P if the last state in the corresponding trace of this automaton is an accept state: $s_n \in A$. An example of an FSA is given in Figure 1. This FSA has two states s_0 and s_1, and so $S = \{s_0, s_1\}$. State s_0 is the start state, which is denoted by an arrow with no origin. The alphabet of this FSA is $\{\texttt{open}, \texttt{close}, \texttt{C}\}$. There is a transition from state s_0 to state s_1 based on event `open`. Graphically, we may represent several transitions from state s to state s' with a single arrow that is labeled with a list of events on which all these transitions are based. For example, in Figure 1 the self-arrow on state s_1 is labeled `open, C` and thus represents two transitions, $\delta(s_1, \texttt{open}) = s_1$ and $\delta(s_1, \texttt{C}) = s_1$. This FSA accepts the sequence `open, C, close`, because it has a trace s_0, s_1, s_1, s_0 on this sequence and the last state in this sequence, s_0, is an accept state, as denoted

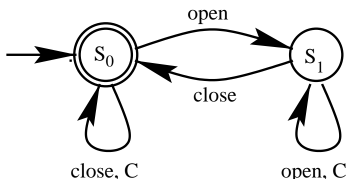

Figure 1: An example FSA or Büchi automaton

by concentric circles.

We call the set of properties that can be specified as an FSA *regular event sequencing* properties. Such a property holds for a system if for any terminating execution of this system the sequence of events observed on this execution puts the FSA in an accept state.

3.2. Overview of the *FE* Approach

FLAVERS models the software system under analysis as a *Trace Flow Graph (TFG)*. The TFG is based on the control flow graphs (CFGs) for the components of the system, where the nodes in the TFG may be labeled with events. We call the collection of all events with which the nodes of the TFG are labeled the *alphabet* of this TFG. To reduce the size of the representation, the CFGs are refined to remove all nodes that are not labeled with an event or that do not affect the sequencing of events. Thus, the resulting refined CFGs correctly capture all possible sequences of events associated with their corresponding component. At present, FLAVERS handles interprocedural systems by in-lining called routines. Since nodes with events are usually a small subset of all the nodes in the original CFG, the refined CFG is typically much smaller than the original CFG. Thus, in our experience, in-lining of refined CFGs usually does not cause a severe blow-up in the size of the CFG representation.

The TFG for a concurrent system is obtained by connecting the refined, in-lined CFGs for all threads of control with additional nodes and edges. Unique *initial* and *final* nodes represent the start and the end states of the system respectively. In addition, depending on the concurrency semantics of the system being modeled, the TFG may include special nodes that represent communication among the threads of control. In all cases, special edges that represent interleavings of events from the threads of control executing in parallel are added to the TFG. Each path from the initial to the final node in the TFG represents a sequence of events that occur on the nodes along this path. The TFG is a *conservative* representation of the sequences of events that could occur along a system execution. That is, any sequence of events in the TFG alphabet that could occur during execution of the system is represented by some path in the TFG with a corresponding event sequence. However, the converse is not true, since CFGs and thus TFGs may contain a number of *infeasible paths*, which do not correspond to any system executions.

Formally, a TFG is a labeled directed graph $G = (N, E, n_{initial}, n_{final}, \Sigma_G, L)$, where N is the set of graph nodes, E is the set of edges, $n_{initial} \in N, n_{final} \in N$ are the initial and final nodes, Σ_G is an alphabet of event labels associated with the graph, and $L : N \rightarrow \Sigma_G$ is a function that labels some of the nodes of the graph with an event drawn from this alphabet. For convenience in introducing the algorithm, with each node n in the TFG we associate a set $Pred(n)$ containing all predecessors of n.

A property specified as an FSA holds for a system if this FSA accepts event sequences for all paths through the TFG for this system. FLAVERS uses the data flow based *FE* analysis algorithm to solve this problem. This is done by associating states of the property FSA with the nodes of the TFG. We use a forward flow data flow algorithm where states are propagated from one node to another, depending on the FSA transition function associated with the events that are encountered in the TFG. Thus, a state s is associated with node n if and only if there is a path from the initial node of the TFG to n that encounters a sequence of events that drives the property FSA to state s when the path reaches n. Note that since multiple paths may exist from the initial node to node n, a set of property states may be associated with each node. The iterative worklist algorithm continues to propagate states to nodes in the TFG until it reaches a fixed point, where no additional states can be associated with TFG nodes. The outcomes of this analysis are either that (1) the set annotating the final node of the TFG contains only accept states of the FSA, indicating that the property holds on **all** executions of the system or (2) the set annotating the final node of the TFG contains at least one non-accept state of the FSA, which means that the property **may** not hold on **some** executions.

The alphabet of the property must be a subset of the events in the alphabet of the TFG: $\Sigma \subset \Sigma_G$. To represent the fact that the property "ignores" the events not in Σ, we can modify the transition function of the FSA to contain self-transitions on all states of the property for all events that are in the TFG alphabet but not in the FSA alphabet: $\forall s \in S, \forall e \in \Sigma_G \setminus \Sigma, \delta(s, e) = s$. As a result of this modification, the alphabet of the property becomes equal to the alphabet of the TFG: $\Sigma = \Sigma_G$.

If the analysis finds that a property holds on all paths through the TFG, then it is guaranteed to hold on all possible executions of the system. When the analysis indicates that the property does not hold on some

paths through the TFG, this may be because the system is in error or it may be because all the paths in the system model that violate this property correspond to infeasible paths. FLAVERS provides a means for selectively removing infeasible paths from consideration by allowing the analyst to add *feasibility constraints*, finite state automata that model semantic restrictions on the system's execution that are not reflected in the TFG. For example, CFGs, and the TFGs constructed from them, typically do not model the values assigned to variables during execution. Thus, paths through the TFG may not represent feasible executions because these paths do not respect the values of some variables. A feasibility constraint could be constructed to track the possible finite values or ranges of values of such a variable, thereby eliminating some or all infeasible paths. Formally, a constraint automaton is an FSA $C = (S_C, s_C, \Sigma_C, \delta_C, c_C)$, where c_C is a unique *crash* state.

Each feasibility constraint has a distinct crash state, which signifies that the sequence of events applied to the constraint does not correspond to any legal behavior of the system. The crash state of a constraint indicates that this constraint is violated. For any state $t \in S_C$ and any event $e \in \Sigma_C$, $\delta_C(t, e) = c_C$ if and only if observing event e at state t does not correspond to any legal behavior of the constraint. The crash state is a sink, which means that there are no transitions from this state to any other state in the constraint. When feasibility constraints are used, the *FE* algorithm propagates tuples of states where each tuple has an element that represents a state of the property and an element for a state of each of the constraints. More precisely, *tuple* $T = (p, c_1, ..., c_k)$, where $p \in S_P, c_i \in S_{C_i}, \forall 1 \leq i \leq k$ and the *start* tuple T_0 is the tuple $(s_P, s_{C_1}, ..., s_{C_k})$. If one of the elements in a tuple represents a crash state for a constraint, this tuple is not propagated beyond this node.

Similar to the case where no feasibility constraints are used, the *FE* algorithm runs until it reaches a fixed point, after which the states of the property annotating the final TFG node determine whether the property holds on all executions of the system.

In the following, we refer to the collection consisting of the TFG, property automaton P, and the constraint automata $C_1, ..., C_k$ as an *analysis problem*. We require that the alphabets of the property and all constraint automata be contained in the alphabet of the TFG: $\Sigma_P \subseteq \Sigma_G, \forall i, 1 \leq i \leq k, \Sigma_{C_i} \subseteq \Sigma_G$.

We refer to the collection of all possible tuples for a given analysis problem as *Tuples*:

$$Tuples = \bigcup_{p \in S_P} \bigcup_{c_1 \in S_{C_1}} \cdots \bigcup_{c_k \in S_{C_k}} (p, c_1, ..., c_k)$$

A *tuple transition function* $\Delta : N \times Tuples \rightarrow Tuples$ describes propagation of tuples through TFG nodes. It is defined as follows:

$$\forall n \in N, T = (p, c_1, ..., c_k) \in Tuples,$$
$$\Delta(n, T) = (\delta_P(p, L(n)), \delta_{C_1}(c_1, L(n)), ..., \delta_{C_k}(c_k, L(n)))$$

3.3. The *FE* Algorithm

The *FE* algorithm of FLAVERS is a forward flow data flow algorithm over the TFG with the power-set of *Tuples* as the lattice. The function space is provided by function Ω :
$2^{Tuples} \times N \rightarrow 2^{Tuples}$ based on the tuple transition function Δ:

$$\begin{aligned} &\forall n \in N, A \in 2^{Tuples}, \\ &\Omega(A, n) = \{T \mid \exists T' \in A, \Delta(T', n) = T\} \end{aligned} \tag{1}$$

The *FE* algorithm associates two sets with each node n in the TFG, $IN(n)$ and $OUT(n)$. The *IN* set for node n represents the possible states of the system immediately before this node is executed. This set is computed as the union of all possible states in which the system can be after the predecessor nodes for n are executed:

$$IN(n) = \bigcup_{p \in Pred(n)} OUT(p) \tag{2}$$

The *OUT* set for node n represents the possible states of the system immediately after this node is executed. This set is computed by applying the transition function to n and the tuples in its *IN* set and removing from the result all tuples that contain at least one constraint crash state:

$$\begin{aligned} OUT(n) = &\Big(\bigcup_{T \in IN(n)} \Delta(n, T) \Big) \setminus \{T = (p, c_1, ..., c_k) \in \\ &Tuples \mid \exists i, 1 \leq i \leq k, c_i = c_{C_i}\} \end{aligned} \tag{3}$$

The algorithm is initialized by setting the *OUT* set of the initial TFG node to contain the start tuple and setting all other *IN* and *OUT* sets to be empty.

The algorithm repeatedly recomputes *IN* and *OUT* sets of the TFG nodes in an arbitrary order, until a fixed point is reached. To determine if the property holds on all terminal executions of the system, all tuples in the *OUT* set of the final TFG node are investigated. The property holds if all states of the property FSA in these tuples are accept states: $\forall T = (p, c_1, ..., c_k) \in OUT(n_{final}) : p \in A_P$. If this condition is not true, FLAVERS concludes that the property does not hold.

4. The *SIE* Version of FLAVERS

By making a simple modification to the *FE* algorithm, FLAVERS can check safety properties on infinite executions. The most important change is in the representation of properties. Although we still use FSAs to represent safety properties to be checked on infinite executions, these FSAs have somewhat different semantics from those used in the *FE* algorithm.

Any event sequencing safety property can be formulated in a form that describes undesirable behaviors of the software system under analysis. The reason for this is that safety properties are finitely refutable statements and so they can be represented as sequences of events that should be observed to refute the property. The refutation event must be explicit and thus represents a certain point in the event sequences. Similar to the approach of [6], in FSAs modeling safety properties, we define a special *violation* state v, which represents the property being refuted. v is a sink state, which means that there is a transition from v to v on any event in the alphabet of this FSA. We say that sequences of events that correspond to traces of this FSA that contain the violation state v *violate* the safety property represented by this FSA. The following theorem offers a proof that any regular event sequencing safety property can be represented as an FSA with a violation state.

Theorem 1. *Any regular event sequencing safety property can be represented as an FSA with a unique violation state v, such that the property does not hold on an event sequence if and only if the trace of the FSA corresponding to this sequence contains v.*

The converse is true as well: any FSA with a violation state represents a safety property.

Proof. Due to the space limitations, we do not present the full formal proof, which is based on the formal definition of safety [1].

□

The *SIE* algorithm of FLAVERS uses FSAs with a violation state to represent properties. This algorithm proceeds in exactly the same way as the *FE* algorithm, with the exception that instead of checking the final node of the TFG for violations, we check all nodes. It is not sufficient to check only the final node, because it represents the terminal state of the system, in which all threads of control terminated. This terminal state is never reached if at least one thread enters an infinite loop. Thus, the *SIE* algorithm checks if any node n of the TFG contains a tuple T such that the property in this tuple is in the violation state; this represents a violation of the property.

5. The *LIE* Version of FLAVERS

In this section we first describe the representation of liveness properties used in our *LIE* FLAVERS algorithm, present an overview of the approach, and then give the details of the *LIE* algorithm itself.

5.1. Representing Liveness Properties

Since FSAs can encode only finite event sequences, we need a different formalism to describe infinite behaviors. ω-automata [21] provide such a formalism. Usually, for an infinite trace sequence σ to be accepted by an ω-automaton, some infinite pattern of accept states of this automaton must be observed on the traces of this automaton on σ. In particular, we use a well-known subclass of ω-automata, Büchi automata. A *deterministic* Büchi automaton is an automaton $(S_B, s_B, \Sigma_B, \delta_B, A_B)$, where S_B is a set of states, $s_B \in S_B$ is a start state, Σ_B is the alphabet, A_B is a set of accept states, and δ_B is the transition function $S_B \times \Sigma_B \rightarrow S_B$. A trace of a Büchi automaton on an infinite event sequence $\sigma = e_1, e_2, ...$ is an infinite sequence of states $s_0, s_1, ...$, where s_0 is the start state and for any $i \geq 1$, there is a transition from s_{i-1} to s_i on event e_i. A Büchi automaton accepts an infinite sequence of events $e_1, e_2, ...$ if the corresponding trace contains an infinite number of accept states. For example, the automaton in Figure 1 can be viewed as a Büchi automaton. An infinite event sequence of alternating `open` and `close` events `open`, `close`, `open`, `close`, ... is accepted by this automaton because the corresponding trace of this automaton $s_0, s_1, s_0, ...$ contains an infinite number of occurrences of accept state s_0.

An arbitrary Büchi automaton cannot be used as a liveness property in our approach. The reason for this is that in in FLAVERS formulation, if the event associated with a node is not in the alphabet of a (property or constraint) automaton, the automaton does not change state when the transition function for tuples is used to compute the *OUT* set for this node. Thus, it is possible that an execution trace has a suffix in which all tuples have the Büchi property automaton in its accept state because none of the events in this trace is in the alphabet of this property. To avoid this complication, we modify each Büchi property automaton in a way that makes its alphabet equal to the alphabet of the TFG. The modification is based on creating an additional non-accept state for each accept state in the Büchi automaton and having transitions on events that are not in the alphabet of the Büchi automaton go from each accept state to its newly created non-accept state.

The precise modification is given here. Let A be a Büchi automaton for which $\Sigma_A \subset \Sigma_T$, where Σ_T is the alphabet of the TFG. We build a new Büchi automaton A'

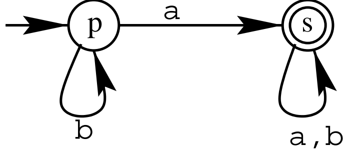

Figure 2: An example property before transformation

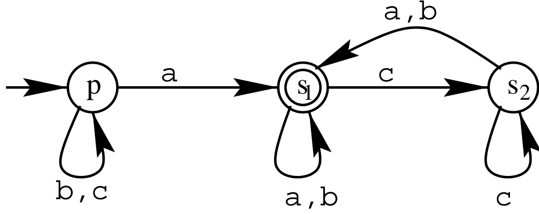

Figure 3: An example property after transformation

equivalent to A in the sense that it accepts the same set of infinite strings in the following way:

1. Copy A to A'.

2. Set $\Sigma_{A'} = \Sigma_T$.

3. For each non-accepting state $s \in S_{A'}$, create new transitions $\delta_{A'}(s, e) = s$ for each $e \in \Sigma_T \setminus \Sigma_A$.

4. For each accepting state $s \in S_{A'}$, split it into two separate states s_1 and s_2, where s_1 is an accepting state and s_2 is a non-accepting state. Let U be the set of transitions entering s from other states of A', V the set of self-transitions on s, and W the set of transitions from s to other states of A'.

   We replace s with s_1 and s_2 as follows. $\forall t \in U$, create t' from the state from which t originates to s_1. $\forall t \in V$, create a self-transition on s_1. $\forall t \in W$, create a transition from s_1 to the state that t is going to. In addition, $\forall e \in \Sigma_T \setminus \Sigma_A$, create a transition based on e from s_1 to s_2. For s_2, create self-transitions on all events in $\Sigma_T \setminus \Sigma_A$, $\forall t \in W$, create a transition from s_2 to the state that t is going to, and $\forall t \in V$, create a transition based on the event that t is based on from s_2 to s_1.

Note that after the modification, $\forall e \in \Sigma_A, \exists t \in V \cup W$: t is based on e, but $\forall e \in \Sigma_T \setminus \Sigma_A, \neg\exists t \in V \cup W : t$ is based on e. Figures 2 and 3 illustrate the Büchi automaton transformation defined here. We assume that $\Sigma_T = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ and $\Sigma_A = \{\mathsf{a}, \mathsf{b}\}$. Figure 2 shows a Büchi automaton before the transformation and Figure 3 shows the corresponding Büchi automaton after the transformation. Note that this transformation does not cause a severe blow-up in the size of the Büchi property automaton, since the number of extra states created equals the number of accept states in the original Büchi property automaton.

5.2. Overview of the *LIE* Approach

A direct and very naive approach to checking properties on infinite executions would be to follow the *FE* algorithm with a Büchi automaton representing the property of interest but preserve the history of changes for each (T, n) pair, starting with the initial pair $(T_0, n_{initial})$. Then we can check if the current state is already present in this history and if an accept state of the property has been entered since its last occurrence. Of course, the complexity of storing and perusing all that additional history information is prohibitive. Instead, we use the *FE* algorithm but then evaluate the TFG with all the tuples assigned to its nodes to find infinite behaviors. In the rest of this chapter we give the details of this *LIE* algorithm and the artifacts that it relies upon.

As described in Section 3.3, the *FE* algorithm of FLAVERS associates sets of tuples $OUT(n)$ with each node n in the TFG. A tuple T is in $OUT(n)$ if there is a path through the TFG from the initial node $n_{initial}$ to n that corresponds to a trace of events that would cause the automata for the property and all constraints to transfer from their start states to the states represented by tuple T. Thus, the problem of determining whether a particular tuple T appears in the OUT set of node n can be viewed as a reachability problem in the *tuple-node* space $(Tuples, N)$. Formally, the tuple-node space $(Tuples, N)$ is a structure (P, E_{tn}), where P is the set of pairs (T, n) such that $T \in OUT(n)$ after the *FE* state algorithm terminates and E_{tn} is the set of edges, where $((T_1, n_1), (T_2, n_2)) \in E_{tn}$ if $(T_1, n_1), (T_2, n_2) \in P \wedge n_1 \in Pred(n_2) \wedge \Delta(n_1, T_1) = T_2$.

We say that there exists a *path* from pair (T, n) to pair (T', n') if there are pairs $(T_1, n_1), ..., (T_k, n_k)$ for some $k \geq 0$, such that $((T, n), (T_1, n_1)), ((T_1, n_1), (T_2, n_2)), ...,$ $((T_k, n_k), (T', n')) \in E_{tn}$. A *reachability function Reach* : $P \rightarrow 2^P$ for a given pair returns the set of all pairs that can be reached for this pair through a path in the tuple-node space: $\forall (T, n) \in (Tuples, N), Reach((T, n)) = \{(T', n') | \exists$ a path from (T, n) to $(T', n')\}$.

From an abstract level, our *LIE* algorithm uses an approach for analyzing the tuple-node space that is very similar to the approach used by model checking and reachability analysis approaches. We attempt to identify strongly-connected components in the tuple-node space that do not have tuples containing an accept state of the property. If such a strongly-connected component is found, it represents one or more infinite executions on which an accept state of the property is

not entered infinitely often. By the definition of Büchi automata acceptance, the property is violated on such executions. On the other hand, the absence of such strongly-connected components signifies that the liveness property holds on all infinite executions of the program. In the rest of this section we describe this algorithm in detail.

5.3. The *LIE* Algorithm

The following algorithm for checking liveness properties with FLAVERS assumes that the *FE* algorithm is used first, and so every node of the TFG has a set of of tuples OUT associated with it. The following steps are then performed:

1. Remove from the OUT sets of all TFG nodes all tuples where the Büchi automaton is in an accept state.
2. Find all maximal strongly-connected components in the resulting (reduced) tuple-node space. A maximal strongly-connected component in the tuple-node space $(Tuples, N)$ is defined as a set of tuple-node pairs $C \subseteq P$ such that
   (a) $\forall (T_1, n_1), (T_2, n_2) \in C$,
   $(T_2, n_2) \in Reach((T_1, n_1))$ and
   (b) $\forall (T_1, n_1) \in C, (T_2, n_2) \in P \setminus C$,
   $(T_1, n_1) \notin Reach((T_2, n_2)) \vee$
   $(T_2, n_2) \notin Reach((T_1, n_1))$.
3. If at least one strongly connected component has been found, the property is violated. This property violation can be illustrated by inserting in the OUT sets of all TFG nodes the tuples that were removed in step 1 of this algorithm and showing a path from $(T_{initial}, n_{initial})$ to this strongly-connected component.

Intuitively, if after removing all tuples in which the property automaton is an accept state, no strongly-connected components exist in the tuple-node space, it means that no execution can be found on which the property automaton enters an accept state only a finite number of times. This means that the liveness property being checked holds on all possible program executions. Alternatively, if a strongly connected component is found, it represents a suffix of an infinite execution such that on this suffix no accept states of the property are entered. Thus, on this execution an accept state of the property is entered only a finite number of times, and so the property is violated.

This approach is similar to the one used by model checking [9]. In fact, our approach can be reduced to checking a specific CTL formula $AGAFa$ with model checking, where a is true in a tuple-node pair (T, n) if and only if the state of A in T is accepting. The major difference between our approach and that of model checking is in the way that the state space of the system is represented and in the way this representation is computed.

5.4. Properties of the *LIE* Algorithm

We need to prove termination, conservativeness, and a statement about complexity of this algorithm.

Theorem 2 (Termination). *For any LIE analysis problem $(G, P, C_1, ..., C_k)$, the algorithm terminates.*

Proof. This follows from the fact that the tuple-node space is finite and termination of the efficient Tarjan algorithm [20] for computing maximal strongly-connected components. □

Conservativeness of our algorithm means that if there exists an execution of the system on which the Büchi property automaton does not hold, the algorithm will detect that.

Theorem 3 (Conservativeness). *If there exists an execution of the system on which there is a suffix where an accept state of the property Büchi automaton is not reached infinitely often, our algorithm will detect that.*

Proof. Suppose that there is an execution of the system with a suffix on which the Büchi automaton never enters an accept state. Since both the TFG model and the *FE* algorithm are conservative [11], this means that there is a trace through the tuple-node space of the problem on which the Büchi automaton never enters an accept state. Since the tuple-node space is finite, this trace must correspond to a loop L in the tuple-node space. When our algorithm eliminates all states of the tuple-node space that correspond to tuples in which the Büchi property automaton in an accept state, loop L is still present, since in no tuples along this loop is the property in an accept state. Thus, there exists a strongly-connected component that contains this loop, and so our algorithm will conclude that the property may be violated. □

The following theorem states the worst-case complexity of the algorithm.

Theorem 4 (Worst-case Complexity). *The worst-case complexity of our algorithm as described is $\mathcal{O}(|N|^2|Tuples| + |E_{tn}|)$.*

Proof. The $|N|^2|Tuples|$ component of the complexity formula in the statement of this theorem is just the worst-case complexity of the *FE* algorithm that must be done first. The worst-case complexity of the Tarjan

algorithm for finding all maximal strongly-connected components of the tuple-node space is $\mathcal{O}(|P| + |E_{tn}|)$. By observing that $|P| \leq |N||Tuples|$, we arrive at the stated complexity. □

This worst-case result is consistent with the complexity of other finite state verification approaches on liveness properties, except for [10], where the worst-case bound in general cannot be expressed in terms of the characteristics of the property and system models.

5.5. Implementation

We have implemented the approach proposed in this paper and carried out an initial, very preliminary, experiment, in which we dealt with two liveness properties for a concurrent Ada producer/consumer example. In this example, multiple producer threads put items in an unbounded buffer and multiple producer threads extract items from this buffer. Our first property specifies that a consumer thread does not starve, i.e. on all infinite executions a consumer thread extracts items from the buffer an infinite number of times. This property can be violated, since the example does not guarantee fair treatment of all threads. Our implementation correctly finds an infinite execution that demonstrates starvation of a consumer thread. Our second property specifies that on all infinite executions some buffer activity (putting or extracting items) happens infinitely often. Our implementation correctly demonstrated that this property holds on all possible executions of the example.

The producer/consumer example is scalable; we checked the two properties described above on four different sizes of the example: 2, 4, 6, 8, where the size corresponds to the number of consumers/producers in the example. (Thus, the example of size 2 has two producers and two consumers.) For each of the sizes the outcome described in the previous paragraph was obtained. An interesting observation is that for both properties, the number of required constraints did not depend on the size of the example. For the first property we needed two constraints modeling control flow through select threads and for the second property we needed three similar constraints. In all cases, checking each of the properties took under 4 seconds on a Pentium III Xeon 550 MHz machine.

6. Fairness Assumptions and Precision Improvements

To be conservative, FLAVERS assumes that all traces through the TFG or tuple-node space correspond to executable behavior in the system being analyzed. Constraints can be used to eliminate infeasible traces selectively. For infinite executions, the algorithm described above assumes that all loops can be executed infinitely. It would be more realistic to recognize that some loops can execute infinitely, while others cannot. Program optimization techniques could be used to statically detect at least some of the finite loops. Using FLAVERS constraint mechanism (e.g. modeling values of variables used in loop predicates), information could be provided to improve or refine this static analysis. Alternatively, we believe that it may be more practical to let the analyst mark those loops in threads that can never execute infinitely (or, the analyst may mark all potentially infinite thread loops). Given this information, the above algorithm can be modified so as not to consider the strongly-connected components in the tuple-node space that correspond to a set of loops in the control flow of individual threads, if any of these loops cannot be infinite.

Fairness conditions are often employed to ensure that some reasonable behaviors of a system are taken into account. For example, in a client-server configuration of system threads, a possible fairness requirement is that if two client threads request a service S infinitely often and the server satisfies S infinitely often, then both clients obtain the service infinitely often (in other words, it is not possible for one of the clients to "starve" while the other always gets the service). With FLAVERS, we can again use the feasibility constraint mechanism to represent fairness assumptions. Because feasibility constraints are FSAs, these assumptions are rather strong. For example, using only FSAs, it is impossible to represent the fairness assumption about the client-server system described above. However, we can represent an assumption that after client A requested service, the server can serve at most 3 requests from clients other than A before serving client A. An FSA modeling this fairness assumption is shown in Figure 4. Transitions labeled `request A` represent the event of client A requesting service and transitions labeled `serve A` and `serve other` represent the events of the server serving A and a client other than A respectively. (Note that in this example, we make two reasonable assumptions about the system: (1) a client does not post a request if it has one unsatisfied request outstanding and (2) the server does not provide an unrequested service.) We believe that such fairness conditions are practical, since they can be derived from the actual specifications of the description of the environment in which the software system under analysis has to execute, unlike fairness conditions that specify that a service will be offered infinitely often.

7. Conclusions and Future Work

In this paper, we have extended the original data flow analysis algorithm of FLAVERS (*FE* algorithm) to

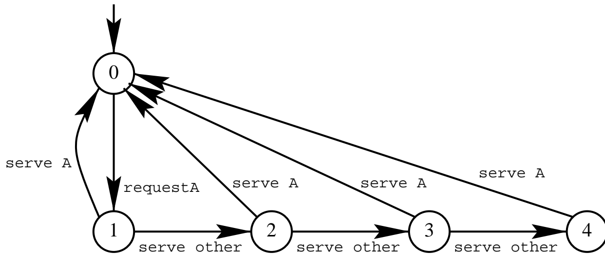

Figure 4: A fairness FSA example

check properties on infinite executions of concurrent software systems. Two different algorithms are presented, one for checking safety properties and the other for checking liveness properties. Although, by representing safety properties as Büchi automata, we could use the *LIE* algorithm for checking both kinds of properties on infinite executions, the *SIE* algorithm has better worst-case complexity bounds than the *LIE* algorithm. Both of these algorithms do not involve changing the existing analysis algorithm of FLAVERS but rather add to it, in a language independent way. This means that the feasibility constraints of FLAVERS that improve precision of the analysis can be used successfully with the proposed algorithms. This is particularly attractive since feasibility constraints can be used to model fairness assumptions about the system under analysis or to refine information about infinite and finite loops. Of course, the problem of determining precisely whether a given loop can be infinite is undecidable. Efficient, conservative automated techniques can be used for this problem and supplemented with guidance from the analyst. With such information, the precision of the analysis results would improve considerably. Thus, we believe that this approach would provide a more precise and realistic basis for analysis and incorporates application-specific fairness and executability considerations.

The worst-case complexity of the *SIE* algorithm is the same as that of the *FE* algorithm, $\mathcal{O}(N^2S)$, where N is the number of nodes in the model of the software system under analysis, S is the number of states in the synchronous cross-product of the automaton representing the property of interest and all feasibility constraint automata used by FLAVERS to improve its analysis precision. The worst-case complexity of the proposed algorithm is $\mathcal{O}(N^2S + E)$, where E is the number of transitions among the states of the cross-product automaton. This complexity is similar to that of other finite state verification approaches for checking liveness properties of concurrent software [9, 13].

References

[1] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, Oct. 1985.

[2] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[3] G. R. Andrews. *Concurrent Programming — Principles and Practice.* Benjamin/Cummins Publishing Company Ltd., 1991.

[4] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, May 1985.

[5] S. C. Cheung, D. Giannakopoulou, and J. Kramer. Verification of liveness properties using compositional reachability analysis. In *Proceedings of the 6th European Software Engineering Conference and 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 227–243, Sept. 1997.

[6] S. C. Cheung and J. Kramer. Tractable dataflow analysis for distributed systems. *IEEE Transactions on Software Engineering*, 20(8):579–593, Aug. 1994.

[7] S. C. Cheung and J. Kramer. Compositional reachability analysis of finite-state distributed systems with user-specified constraints. In *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 140–151, Oct. 1995.

[8] S. C. Cheung and J. Kramer. Checking safety properties using compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 8(1):49–78, Jan. 1999.

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

[10] J. C. Corbett and G. S. Avrunin. Using integer programming to verify general safety and liveness properties. *Formal Methods in System Design*, 6:97–123, Jan. 1995.

[11] M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 62–75, Dec. 1994.

[12] M. B. Dwyer and L. A. Clarke. Flow analysis for verifying specifications of concurrent and distributed software. Technical Report 1999-52, University of Massachusetts, Amherst, Aug. 1999. `ftp://ftp.cs.umass.edu/pub/techrept/techreport/1999/UM-CS-1999-052.ps`.

[13] G. J. Holzmann. The model checking SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[14] G. E. Hughes and M. J. Creswell. *Introduction to Modal Logic*. Methuen, London, 1977.

[15] R. Milner. *A Calculus of Communicating Systems*, volume 92. Springer-Verlag, Berlin, 1980.

[16] G. Naumovich, G. S. Avrunin, and L. A. Clarke. Data flow analysis for checking properties of concurrent Java programs. In *Proceedings of the 21st International Conference on Software Engineering*, pages 399–410, May 1999.

[17] G. Naumovich, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil. Applying static analysis to software architectures. In *Proceedings of the 6th European Software Engineering Conference and 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 77–93, Nov. 1997.

[18] K. M. Olender and L. J. Osterweil. Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE Transactions on Software Engineering*, 16(3):268–280, Mar. 1990.

[19] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Symposium on Foundations of Computer Science*, pages 46–57, Oct.–Nov. 1977.

[20] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972.

[21] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 4, pages 133–191. Elsevier Science Publishers B. V., 1990.

[22] M. Y. Vardi. Verification of concurrent programs: The automata-theoretic framework. In *Proceedings of the 2nd Annual Symposium on Logic in Computer Science*, pages 167–176, June 1987.