Automated Systematic Testing for Constraint-Based Interactive Services

Patrice Godefroid, Lalita J Jagadeesan
Software Production Research Dept
Bell Laboratories, Lucent Technologies
Naperville, IL
{god,lalita}@research.bell-labs.com

Radha Jagadeesan, Konstantin Läufer
Dept. of Mathematical and Computer Sciences
Loyola University Chicago
Chicago, IL
{radha,laufer}@cs.luc.edu

ABSTRACT
Constraint-based languages can express in a concise way the complex logic of a new generation of interactive services for applications such as banking or stock trading, that must support multiple types of interfaces for accessing the same data. These include automatic speech-recognition interfaces where inputs may be provided in any order by users of the service. We study in this paper how to systematically test event-driven applications developed using such languages. We show how such applications can be tested automatically, without the need for any manually-written test cases, and efficiently, by taking advantage of their capability of taking unordered sets of events as inputs.

Categories and Subject Descriptors
H.4.m [**Information Systems**]: Miscellaneous; D.2 [**Software**]: Software Engineering; D.2.4 [**Software Engineering**]: Software / Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms
Languages, Reliability, Verification

Keywords
Constraint-Based Languages, Verification, Testing, Model Checking, Interactive Services, State Explosion, State-Space Reduction

1. INTRODUCTION
Today, it is becoming more and more commonplace for modern interactive services, such as those for personal banking or stock trading, to have more than one interface for accessing the same data. For example, many banks allow customers to access personal banking services from an automated teller machine, bank-by-phone interface, or web-based interface. Furthermore, telephone-based services are starting to support automatic speech recognition and natural language understanding, adding further flexibility in interaction with the user. When multiple interfaces are provided to the same service, duplication can be a serious problem in that there may be a different service logic (i.e., the code that defines the essence of the service) for every different user interface to the service. Moreover, to support natural language interfaces, services must allow users considerable flexibility in the way they input their service requests. Thus, it is desirable for the programming idioms and methods for such services to provide the following capabilities:

- allow requests to be phrased in various ways (e.g., needed information can be provided in any order),
- prompting for missing information,
- correction of erroneous information,
- lookahead (to allow the user to speak several commands at once), and
- backtracking to earlier points in the service logic.

Constraint-based languages (for a foundational overview, see [16, 19]) provide a suitable paradigm to support the required flexibility in user inputs. Examples of such languages in dialogue management include methods based on frames [1], forms [9, 15], approaches based on AND/OR trees [22] and Sisl (Several Interfaces, Single Logic) [4].

The powerful expressiveness of constraint-based languages allows the construction of succinct programs with complex reactive behavior. This motivates the need for reliable yet cost-effective testing techniques and tools suitable to check the correctness of business-critical applications developed using such languages. We study in this paper how to *automatically* and *efficiently* check temporal properties of programs written in constraint-based languages for interactive services.

We first present a nondeterministic algorithm for systematically testing the logic of a program in a constraint-based language. The possible use of arbitrary host language code and potential elaborate access to external data (such as database lookups) in a full program makes the use of classical constraint satisfaction and model checking techniques problematic. Consequently, this algorithm assumes, as is usually done in testing, that the user specifies a fixed set of possible data values for each user input. The algorithm then dynamically detects at run-time the set of input events that an application is currently ready to accept, and uses that information to drive the execution of the application by sending it inputs selected nondeterministically. Used in conjunction with the systematic state-space exploration tool VeriSoft [11], which supports a special system call (called VS_toss) simulating nondeterminism, this nondeterministic test driver can systematically generate all possible behaviors of any such program. These behaviors can then be monitored and checked against user-specified safety properties. This algorithm thus eliminates the need for manually-written test cases, and is automatically applicable to any program. This testing technique is made possible by the structured interface that an event-driven constraint-based program provides to its environment.

Unfortunately, the expressiveness of constraint-based languages makes this algorithm quite inefficient. For instance, a service that awaits 10 inputs from the user (in no particular order) has 10! different paths to collect the inputs. However, the flip side of this ability to accept unordered sets of inputs that can be provided in any order, is the *inability* of constraint languages to (observationally) distinguish permutations of the unordered sets of inputs. Thus, generating all these sequences is not always necessary to check the temporal properties that we are considering. Our second algorithm makes systematic testing significantly more efficient by exploiting this observation and taking advantage of a form of symmetry induced by constraints and their ability to specify *sets* of input events rather than single events.

While the basic problems and ideas that we discuss in this work are common to any deterministic constraint-based language in our domain of interest, our concrete presentation is in the context of the Sisl language mentioned earlier. Sisl includes a deterministic constraint-based domain-specific language for developing event-driven reactive services. It is implemented as a library in Java. Sisl supports the development of services that are shared by multiple user interfaces including automatic speech recognition based or text-based natural language understanding, telephone voice access, web, and graphics based interfaces. Sisl is currently being used to prototype a new generation of call processing services for a Lucent Technologies switching product.

This paper is organized as follows. In the next section, we describe the Sisl language and formalize its semantics. Then, we present algorithms for automatically and efficiently checking safety properties of Sisl programs. These algorithms have been implemented and implementation issues are discussed in Section 4. A simple example of Sisl application is presented in Section 5. Results of experiments with this example are then discussed. Finally, we present concluding remarks and compare our results with other related work.

2. SISL: SEVERAL INTERFACES, SINGLE LOGIC

2.1 Overview of Sisl

Sisl applications consist of a single service logic, together with multiple user interfaces. The service logic and the interfaces communicate via events: the user interfaces collect information from the user and send it to the service logic in the form of events. The service logic reacts to the events, and reports to the user interfaces its set of enabled events; i.e. the set of events it is ready to accept in its current state. The user interfaces utilize this information to appropriately prompt the user. Any user interface that performs these functions can be used in conjunction with a Sisl service logic. For example, in the context of an interactive natural language service that browses and displays organizational data from a corporate database, Sisl may inform the natural-language interface that it can accept an organization e.g., if the user has said "Show me the size of the organization." The interface may then prompt the user, e.g. by saying "What organization do you want to see?" If the user responds with something that the natural-language interface recognizes as an organization, the interface will then generate a Sisl event indicating the organization was detected. This event will then be processed by the Sisl service logic.

In Sisl, the service logic of an application is specified as a *reactive constraint graph*, which is a directed graph with an enriched structure on the nodes. The traversal of reactive constraint graphs is driven by the reception of events from the environment: these events have an associated label (the event name) and may carry associated data. In response, the graph traverses its nodes and executes actions; the reaction of the graph ends when it needs to wait for the next event to be sent by the environment. Predicates on events play an important role in reactive constraint graphs. In particular, nodes may contain an ordered set of predicates, which indicate a conjunction of information to be received from the user. Upon receipt of the appropriate events, predicates in the current node are evaluated in order. Satisfaction of all predicates at a node triggers a node change in the graph, as may violation of any predicate.

Reactive constraint graphs are implemented as a Java library. An important aspect of reactive constraint graphs is that nodes may have associated actions, consisting of arbitrary Java code, which is executed upon entry into/exit from the node. These actions can hence have side effects on local and global variables of the Java program, or on external databases. We note that individual predicates do not have actions associated with them.

2.2 The Sisl process algebra

In this section, we describe a process algebra to succinctly describe Sisl programs.

Events. We begin with a description of the events used in the process algebra.

- A set of (input) labels I.
- A set of values $\mathcal{V}$, that are carried on labels from I.

A label with an associated value is called an event, and we write e to range over $[I \times \mathcal{V}]$. A signature is a subset of I. We use σ for a signature, and $|\sigma|$ for its size. A predicate ϕ over a signature σ is a boolean valued function from $\mathcal{V}^{|\sigma|} \rightarrow \texttt{Bool}$; that is, it maps every set $\{(a, v) | a \in \sigma, v \in \mathcal{V}\}$ to $\{true, false\}$. The signature of a predicate ϕ is sometimes written $\sigma(\phi)$.

Syntax. Terms of the process algebra have the following abstract syntax (where P, P_i, $P_{\texttt{next}}$ and $P^i_{\texttt{viol}}$ denote processes).

$$
\begin{array}{rcll}
P & ::= & \texttt{n}ull & \\
 & | & P_1 ; P_2 & \text{(Sequential composition)} \\
 & | & \sum(a_i, v_i).P_i & \text{(Choice)} \\
 & | & \texttt{c}onstraint(P_{\texttt{next}}, \langle(\phi_i, \sigma(\phi_i)), P^i_{\texttt{viol}}\rangle i = 1 \ldots n) & \\
 & & & \text{(Constraint)}
\end{array}
$$

We refer to P^i_{viol} as the target node of predicate ϕ_i.

Informal dynamic semantics. The process combinators have the following intuitive meaning. Sequential composition is standard: "$P_1; P_2$" means that P_1 is executed first and P_2 is executed after P_1 terminates.

Choice, $\sum(a_i, v_i).P_i$ functions like prefixing in process algebras: the process waits for an event with label in the a_i's and with data v_i. Such a term corresponds to a "choice" node in the reactive constraint graph, and represents a disjunction of events to be received from the user. For every event in this set, the Sisl service logic automatically sends out a corresponding prompt to the user. The choice node then waits for the user to send an event in the specified set. When such an event arrives, the corresponding transition is taken, and control transfers to the child node. To ensure determinism of the Sisl program, all events (a_i, v_i) must be distinct.

$\texttt{constraint}(P_{\texttt{next}}, \langle(\phi_i, \sigma(\phi_i)), P^i_{\texttt{viol}}\rangle i = 1 \ldots n)$, the constraint combinator, has the most interesting dynamic behavior. Such a term corresponds to a "constraint" node in the reactive constraint graph, and has an associated ordered set of predicates ϕ_i on events. Intuitively, a constraint node is awaiting *all* the events in $\cup_i \sigma(\phi_i)$. Thus, this node represents a conjunction of information to be received from the user. These events can be sent to the constraint node in any order. When the control reaches the constraint node, the Sisl service logic automatically sends out a prompt event for every event that is still needed in order to evaluate some predicate. In addition, it automatically sends out a optional prompt for all other events mentioned in the predicates — these correspond to information that can be corrected by the user. In every round of interaction, the constraint node waits for the user to send any event that is mentioned in its associated predicates. Each predicate associated with a constraint node is evaluated in order as soon as all of its events have arrived. If an event is resent by the user interfaces (i.e, information is corrected), predicates with that event in their signature are re-evaluated in order.

There are two ways to exit a constraint node.

- All of its predicates have been evaluated and are satisfied. In this case, control transfers to $P_{\texttt{next}}$.
- Some predicate i with non-`null` $P^i_{\texttt{viol}}$ evaluates to `false`, and all predicates j with $j < i$ evaluate to true. That is, predicate i is the first predicate in order that is false, and its $P^i_{\texttt{viol}}$ is non-`null`. In this case, control transfers to $P^i_{\texttt{viol}}$.

As mentioned earlier, node changes may cause side-effecting actions to be executed; however, evaluation, satisfaction, or violation of individual predicates do not cause any side effects to occur, other than the node changes described above.

2.3 Sisl: The state machine semantics

We will describe a associated space of state machines associated with Sisl, and describe the semantics of Sisl processes as state machines.

A state machine in the Sisl semantics is a tuple (I, S, s_0, T, F) where I is the set of input labels, S is a set of states, s_0 is the unique start state, $F \subseteq S$ is the set of final states, and the transition relation $T \subseteq S \times [I \times V] \times S$ is deterministic, i.e., given a state s and any event e, there is at most one s' such that (s, e, s') is in the transition relation. In this case, we write $s \xrightarrow{e} s'$. Final states do not have any outgoing transitions.

Constructions

We will describe the denotations of Sisl combinators as constructions on the state machines. In this section we will use P (perhaps with super/subscripts) to range over the members of the given class of state machines.

The state machine corresponding to `null` has a single state that is both the start state and the final state. It has no transitions.

Sequential composition This is described in a standard fashion by assumption on final states. All arcs into the final states of P_1 are redirected to the unique start state of P_2 in $P_1; P_2$. Formally, the sequential composition of state machines $P_1 = (I_1, S_1, s_1, T_1, F_1)$ and $P_2 = (I_2, S_2, s_2, T_2, F_2)$ is the state machine $P = (I_1 \cup I_2, S_1 \cup S_2, s_1, T, F_2)$ where $T = T_2 \cup \{(s, e, s') \in T_1 | s' \notin F_1\} \cup \{(s, e, s_2) | s \in S_1 \text{ and } (s, e, s') \in T_1 \text{ for some } s' \in F_1\}$

Choice: $\sum(a_i, v_i).P_i$ This is described by building the state machine for P_i, and adding a new start state with transitions labeled (a_i, v_i) to the start state of the state machine for P_i. Formally, given state machines $P_i = (I_i, S_i, s_i, T_i, F_i)$, the resulting state machine P is (I, S, s, T, F), where $I = \bigcup_i I_i$, $S = \bigcup_i S_i$, s is a new state, $F = \bigcup_i F_i$, and $T = (\bigcup_i T_i) \cup \{(s, e, s_i) | e = (a_i, v_i)\}$.

Constraint: $\texttt{constraint}(P_{\texttt{next}}, \langle(\phi_i, \sigma(\phi_i)), P^i_{\texttt{viol}}\rangle i = 1 \ldots n)$

Let $K = \cup_i \sigma(\phi_i)$. The set S of intermediate states the Sisl program goes through while collecting events in K are determined by partial maps $f : K \to V$. Intuitively, the domain of a partial function f, written $\texttt{dom}(f)$, indicates the labels on which data has been received. With this intuition, it is clear that the information still required to satisfy the requirements of this constraint node is given by the transitions that are enabled at this state, and have labels $\{a \mid a \in K, f(a) \text{ undefined}\}$. Furthermore, the start state is given by the partial map with empty domain, since no information has been received.

We write $g = f + (a, v)$ if $g(a) = v$ and f, g are identical on all other labels.

- f is *consistent* if f is such that for all predicates ϕ_i, if $\texttt{dom}(f)$ includes $\sigma(\phi_i)$, then $\phi_i(\langle v_i \rangle) = \texttt{true}$, where $v_i = f(a_i)$ for all $a_i \in \sigma(\phi_i)$.

  f is *inconsistent* otherwise, i.e., there is some predicate ϕ_i such that $\texttt{dom}(f)$ includes $\sigma(\phi_i)$ and $\phi_i(\langle v_i \rangle) = \texttt{false}$, where $v_i = f(a_i)$ for all $a_i \in \sigma(\phi_i)$.
- f is *complete*, if $\texttt{dom}(f) = K$, *incomplete* otherwise.

All four combinations of the above two parameters are possible. Formally, the set S of states is all partial maps $f : K \to \mathcal{V}$, where $K = \cup_i \sigma(\phi_i)$, such that f is either inconsistent or incomplete. Let transition relation T on states in S be defined as follows:

- Let $g = f + (a, v)$ be a consistent and complete map. There is a transition labeled (a, v) from state with label f to $P_{\texttt{next}}$. That is, $P_{\texttt{next}}$ is the target of transitions that make the information contained in a state complete in a consistent fashion.
- Let $g = f + (a, v)$ be an inconsistent map, with ϕ_i being the first i that is falsified in g. If $P_i^{\texttt{viol}}$ is not `null`, then there is an arc labeled (a, v) from state with label f to the start state of $P^i_{\texttt{viol}}$. That is, causing a predicate violation with a non-`null` target node causes control to move to that node.
- There is a transition labeled (a, v) from a state with label f to a state with label $g = f + (a, v)$ if either:
  - g is consistent and incomplete, or
  - g is inconsistent, and $P_i^{\texttt{viol}}$ is `null`, where ϕ_i is the first predicate in order that is falsified in g.

  That is, if no predicates are violated, or if a predicate with a `null` target node is violated, then control remains in the constraint node.

A transition labeled (a, v) adds or changes information on label a at a state. If it changes the information, it is called an *overwrite* event: (a, v) is an overwrite event if it causes a transition from some state f in which $a \in dom(f)$.

The start state is given by the partial map with empty domain. The final states are given by the union of the final states of $P_{\texttt{next}}$ and the non-`null` $P^i_{\texttt{viol}}$'s.

Formally, given state machines $P_{\texttt{next}} = (I_{\texttt{next}}, S_{\texttt{next}}, s_{\texttt{next}}, T_{\texttt{next}}, F_{\texttt{next}})$, and the non-`null` $P^i_{\texttt{viol}} = (I_i, S_i, s_i, T_i, F_i)$, the resulting state machine is $P = (I_P, S_P, s_P, T_P, F_P)$ where $I_P = I_{\texttt{next}} \cup \bigcup_i I_i \cup \bigcup_i \sigma(\phi_i)$, $S_P = S \cup S_{\texttt{next}} \cup \bigcup S_i$, s_P is the partial map with empty domain, $T_P = T \cup T_{\texttt{next}} \cup \bigcup T_i$, and $F_P = F_{\texttt{next}} \cup \bigcup F_i$.

In the following we write $target(\phi_i)$ to refer to the target node of predicate ϕ_i.

3. AUTOMATIC TESTING OF SISL PROGRAMS

In this section, we show how Sisl programs can be automatically and systematically tested for violations of safety properties using *nondeterministic* testing algorithms. Used in conjunction with the systematic state-space exploration tool VeriSoft [11], which supports a special system call "VS_toss" simulating nondeterminism, these nondeterministic testing algorithms can systematically drive the execution of the Sisl application being tested in order to exhibit all its possible behaviors.

Safety properties can be represented by prefix-closed finite automata on finite words [3]. We assume such a representation A_P, and define a safety property $L(P)$ as the set of finite words accepted by the finite automaton A_P.

Let O be the alphabet of A_P. By construction, O is contained in the set of input events I of the state machine M representing the Sisl program to be tested. We call input events in O *observable events*. Let $w_{|O}$ denote the projection of a word $w \in I^*$ over a set $O \subseteq I$.

DEFINITION 3.1. *Let $M = (I, S, s_0, T)$ be the state machine defining the semantics of a Sisl program as defined in the previous section but with $F = S$ (which is therefore omitted). Let $s \overset{w}{\Rightarrow} s'$ denote an execution of M that goes from state s to state s' after receiving a finite sequence w of events which does not include any overwrite events. Let $O \subseteq I$ denote a set of* observable *events. We define the set of* observable behaviors *of M as the language $L_O(M)$ on finite words on O^* such that $L_O(M) = \{w | s_0 \overset{w'}{\Rightarrow} s \wedge w = w'_{|O}\}$.*

In other words, the set of observable behaviors of a Sisl program with respect to a set of observable events O is defined as the set of finite sequences of observable events that the Sisl program can take as inputs excluding overwrite events. For technical convenience and efficiency reasons, we deliberately ignore overwrite events in this definition since their occurrence is an artifact of the user-interface of the system that does not affect transitions from nodes to nodes, and hence the logic of the Sisl program.

The problem we address in this work is thus how to check automatically and efficiently that $L_O(M) \subseteq L(P)$.

A naive solution to this problem consists of driving the execution of the Sisl program by a test driver that nondeterministically sends any enabled input event and associated valid data value to M whenever M is ready to take a new input, the execution of the nondeterministic test driver being itself under the control of VeriSoft. Checking $L_O(M) \subseteq L(P)$ can then be done by monitoring all possible executions of M in conjunction with this test driver, and checking that all its observable behaviors are accepted by A_P. However, this naive approach would generate a state space typically so large that it would render any analysis intractable: for instance, any input event that takes a 32-bit integer as argument would immediately generate a branching point with 2^{32} branches. Clearly, data values are the cause of this unacceptable state explosion.

In the case of constraint nodes for instance, one could think that an analysis of the predicates associated to such nodes using constraint satisfaction techniques might be used to generate automatically data values. However, this approach is problematic in our context since predicates in Sisl programs are implemented by arbitrary Java code, can be quite elaborate and may involve accesses to external data (for instance, the evaluation of a predicate may involve database lookups to fetch and test provisioning data for the subscriber of the service). Also, the evaluation of a predicate on a same set of input data values may change over time (when the evaluation of a predicate may depend on data previously modified during the current execution). How to close automatically any open reactive (Java) program with its most general environment is an interesting but hard problem [5] that is beyond the scope of the present work.

Therefore, we will simply assume here, as is usually done in testing, that the user specifies a fixed set $values(a)$ of possible data values for each input event a in I. Whenever event a is provided as input to the Sisl program during testing, a data value v in $values(a)$ is chosen nondeterministically by the test driver and then passed as argument of event a to the Sisl service.

For a given set V of sets $values(a)$ of data values, we define the restriction of M by V as follows.

DEFINITION 3.2. *Let $M = (I, S, s_0, T)$ be the state machine defining the semantics of a Sisl program as defined in the previous section. Let V be a (complete) valuation function that associates with each input event a a finite nonempty set $values(a)$ of data values: $\forall a \in I : V(a) = values(a)$. We call the* restriction *of M* by *V the state machine $M_V = (I, S, s_0, T_V)$ such that $T' = \{(s, a(v), s') | (s, a(v), s') \in T \wedge v \in V(a)\}$.*

The restriction of M by V is thus the set of states the Sisl program represented by M can reach when data values associated to input events are taken from V exclusively. Note that, since Sisl programs are deterministic, the successor state s' reached after receiving an input event $a(v)$ in a state s is always unique.

In the remainder of this section, we discuss a restricted version of our original problem, namely how to check automatically and efficiently that $L_O(M_V) \subseteq L(P)$, instead of $L_O(M) \subseteq L(P)$.

A simple algorithm for checking whether $L_O(M_V) \subseteq L(P)$ is presented in Figure 1. At any reached state, this algorithm nondeterministically selects an enabled input event a (Step 2.b) and a data value v in $values(a)$ (Step 2.c), and then sends $a(v)$ to the Sisl program being tested (Step 2.d). Nondeterminism is simulated by the special operation "VS_toss" supported by VeriSoft. This operation takes as argument a positive integer n, and returns an integer in $[0, n]$. The operation is nondeterministic: the execution of a transition starting with VS_toss(n) may yield up to $n + 1$ different successor states, corresponding to different values returned by VS_toss.

The execution of this nondeterministic and nonterminating test-driver algorithm is controled by VeriSoft. VeriSoft provides the value to be returned for each call to VS_toss in order to systematically explore all the possibilities. It also forces the termination of every execution when a certain depth is reached. This maximum depth is specified by the user via one of the several parameters that the user can set to control the state-space search performed by VeriSoft, and is measured here by the number of calls to VS_toss executed so far in the current run of the algorithm.

Checking $L_O(M_V) \subseteq L(P)$ can be done as follows. Whenever an event $a(v)$ is sent to the Sisl service SSV to be tested, the automaton A_P representing the property P is evaluated on $a(v)$. If $a \in O$, the automaton may move and reach a new state. By construction, if A_P reaches a non-accepting state, this means that the property P is violated. An error is then reported, and the search stops. The last scenario executed is saved in memory, and can be replayed later by the user with the VeriSoft simulator.

The algorithm of Figure 1 generates all possible sequences of input events (and associated data values) that can be taken as input by the Sisl program. In the rest of this section, we show that generating all these sequences is actually not necessary to check the type of safety properties we consider here. Precisely, we show that the

1. **Initialize the Sisl service SSV.**
2. **Loop forever:**
   (a) **Let E be the set of non-overwrite events in I that are enabled in the current state s. If $E = \emptyset$, halt.**
   (b) **Let i =VS_toss($|E| - 1$); let a be the i^{th} element of E.**
   (c) **Let j =VS_toss($|values(a)| - 1$); let v be the j^{th} element of $values(a)$.**
   (d) **Send $a(v)$ to SSV.**

Figure 1: Simple Algorithm

algorithm of Figure 1 can be optimized so that it does not generate all possible sequences of input events enabled in constraint nodes provided that these events are not observable.

EXAMPLE 3.3. *Let P be a sisl program, and let ϕ_1, ϕ_2, ϕ_3 be any predicates, with signatures given by $\sigma_1, \sigma_2, \sigma_3$. Consider* $\texttt{constraint}(P, \langle\phi_1, \sigma_1, \texttt{null}\rangle, \langle\phi_2, \sigma_2, \texttt{null}\rangle, \langle\phi_3, \sigma_3, \texttt{null}\rangle)$.

Let O, the set of observable events be empty. All interleavings of unobservable input events that drive the system to a same successor node of a constraint node have the same void effect on the property being checked; so, there is no need to generate all of these. Consequently, the testing of this constraint node requires the generation of one *sequence of input events rather than the $(\cup_i \sigma_{\phi_i})!$ sequences generated by the naive algorithm.*

On the other hand, if O is $(\cup_i \sigma_{\phi_i})$, the testing of this constraint node requires the generation of all the $(\cup_i \sigma_{\phi_i})!$ sequences generated by the naive algorithm.

This observation is exploited in the algorithm of Figure 2. This algorithm behaves as the previous one except in the case of constraint nodes. In that case (Step 2.c), the algorithm starts (Step 2.c.ii) by nondeterministically choosing a data value v_a to be associated with each input event a enabled in the constraint node. Then (Step 2.c.iii), it evaluates successively all the predicates ϕ_i of the constraint node. Predicates ϕ_i that are violated by the selected set of data values v_a for each $a \in E$ are added to a set $Marked$ of violated predicates unless there exists another predicate ϕ_j previously added in $Marked$ whose signature $\sigma(\phi_j)$ is contained in the signature $\sigma(\phi_i)$ of ϕ_i, or whose signature $\sigma(\phi_j)$ contains the same set of observable events as $\sigma(\phi_i)$ and $target(\phi_j) = target(\phi_i)$ (Step 2.c.iii.A). If the selected set of data values does not satisfy all the predicates of the node (Step 2.c.iv), one violated predicate ϕ_i in the set $Marked$ such that $target(\phi_i) \neq \texttt{null}$ is nondeterministically chosen, and the input events in its signature are selected in the set S of events to be provided to the Sisl program (Step 2.c.iv.E); otherwise, all predicates are satisfied, and set S is the set of all enabled input events that are enabled in the node (Step 2.c.v). Then, all unobservable input events in set S (if any) are sent to the Sisl program (Step 2.c.vi). Finally, all remaining (hence observable) input events in S are sent to the Sisl program one by one in some random order picked nondeterministically among the set of all possible interleavings of these events (Step 2.c.vii).

The correctness of the algorithm of Figure 2 can be proved by showing that there is a weak bisimulation between the nodes reached during the execution of the algorithm and the nodes of M_V, the restriction of M by V. This in turn guarantees that all observable behaviors of M_V can be observed during the nondeterministic executions of the algorithm of Figure 2. Let $node(s)$ be the current node the Sisl program when it is in state s. We write $n \xrightarrow{w} n'$ to denote that there exists a sequence of non-overwrite input events w' such that $s \overset{w'}{\Rightarrow} s'$, $w = w'_{|O}$, $node(s) = n$, $node(s') = n'$, and no node other than n and n' are traversed during the transition from s to s'.

THEOREM 3.4. *Let $M_V = (I, S, s_0, T_V)$ be the restriction of the state machine M defining the semantics of a Sisl program by a valuation function V. Let $M' = (I, S, s_0, T')$ be the state machine defined with the set T' representing the set of state transitions performed by the Sisl program when it is being tested by the algorithm of Figure 2. Then, for any reachable state s in M_V, we have for $n = node(s)$:*

- *if $n \xrightarrow{w} n'$ in M_V, then $n \xrightarrow{w} n'$ in M'; and*
- *if $n \xrightarrow{w} n'$ in M', then $n \xrightarrow{w} n'$ in M_V.*

PROOF. (Sketch) The proof is immediate if n is not a constraint node. Consider the case where n is a constraint node. Let s be a reachable state in M_V such that $n = node(s)$. To simplify the presentation, assume s is the first state reached when entering node n during that visit of n. (Other cases can be treated in a similar way.) Let us show that every node transition $n \xrightarrow{w} n'$ in M_V can be matched by a node transition $n \xrightarrow{w} n'$ in M' (the converse is immediate).

Since $n \xrightarrow{w} n'$, there exists a sequence of non-overwrite input events w' such that $s \overset{w'}{\Rightarrow} s'$, $w = w'_{|O}$, $node(s) = n$, $node(s') = n'$, and no node other than n and n' are traversed during the transition from s to s'. For simplicity, assume that s' denote the first state reached when entering node n' when executing w' from s. (Again, other cases can be treated in a similar way.)

If n' is the node P_{next} reached when all predicates in n are satisfied, then the set $\{v_a | a \in w'\}$ of data values associated to input events provided during the execution of w' from state s to s' satisfy all the predicates in n. Thus, there exists an execution of the algorithm of Figure 2 that can select this set of data values in Step 2.c.ii. In that case, none of the predicates of node n will be marked, the algorithm will select all the input events in w to be sent to the Sisl program (Step 2.c.v), and there exists one execution of the algorithm that will send all the observable events in w' in the same order as in w (Step 2.c.vii).

Otherwise, n' is the (non-null) target node $target(\phi_i)$ reached after a predicate ϕ_i of n is violated. This means that the set $\{v_a | a \in w'\}$ of data values associated to input events provided during the execution of w' from state s to s' violates predicate ϕ_i. Since $target(\phi_i)$ is reachable, this also means that no other predicate ϕ_j with $j < i$ and such that $\sigma(\phi_j) \subseteq \sigma(\phi_i)$ is violated (otherwise, $target(\phi_j)$ would be reached instead of $target(\phi_i)$ when executing w' from s). There exists an execution of the algorithm of Figure 2 that can select in Step 2.c.ii the set of data values $\{v_a | a \in$

1. **Initialize the Sisl service SSV.**
2. **Loop forever:**
   (a) **Let E be the set of non-overwrite events in I that are enabled in the current state s. If $E = \emptyset$, halt.**
   (b) **If the current node of SSV is NOT a constraint node:**
      i. **Let i =VS_toss($|E| - 1$); let a be the i^{th} element of E.**
      ii. **Let j =VS_toss($|values(a)| - 1$); let v be the j^{th} element of $values(a)$.**
      iii. **Send $a(v)$ to SSV.**
   (c) **If the current node of SSV is a constraint node:**
      i. **Let $\phi_1, \ldots \phi_m$ be the sequence of predicates associated with the constraint node.**
      ii. **Loop through all events a in E (in any order):**
         A. **Let j =VS_toss($|values(a)| - 1$); let v_a be the j^{th} element of $values(a)$.**
      iii. **Loop through all i from 1 to m (in order):**
         A. **If ϕ_i is violated by the data values $\{v_a | a \in E\}$ AND $\forall 0 < j < i : \phi_j \in Marked$ implies**
         $$\sigma(\phi_j) \not\subseteq \sigma(\phi_i) \wedge (target(\phi_i) \neq target(\phi_j) \vee (\sigma(\phi_i) \cap O) \neq (\sigma(\phi_j) \cap O)),$$
         then add ϕ_i to set $Marked$.
      iv. **If $|Marked| > 0$:**
         A. **Remove from $Marked$ all ϕ_i such that $target(\phi_i) = \texttt{null}$.**
         B. **If $|Marked| = 0$: halt.**
         C. **Let i =VS_toss($|Marked| - 1$).**
         D. **Let ϕ_i be the i^{th} element of $Marked$.**
         E. **Let $S = \sigma(\phi_i)$.**
      v. **Else:**
         A. **Let $S = E$.**
      vi. **For all $a \in (S \setminus O)$, send $a(v_a)$ to SSV.**
      vii. **Loop until $(S \cap O) = \emptyset$:**
         A. **Let i =VS_toss($|S \cap O| - 1$); let a be the i^{th} element of $S \cap O$.**
         B. **Send $a(v_a)$ to SSV.**
         C. **Remove event a from set S.**

Figure 2: Optimized Algorithm

$w'\}$. This set of data values violates predicate ϕ_i, which will then be added to the set $Marked$ in Step 2.c.iii of the algorithm, unless there exists another violated predicate ϕ_j with $j < i$, previously added in $Marked$ and such that $target(\phi_i) = target(\phi_j)$ and $(\sigma(\phi_i) \cap O) = (\sigma(\phi_j) \cap O)$; in that case, violating this other predicate ϕ_j also leads to node n' via a sequence w'' of input events such that $w''_{|O} = w$. In any case, a predicate whose violation leads to n' via a sequence w''' of events such that $w'''_{|O} = w$ is added to $Marked$. If this predicate is selected in Step 2.c.iv of the algorithm, all the events in its signature (which includes all events in w) are then sent to the Sisl program, and there exists one execution of the algorithm that will send all the observable events in w''' in the same order as in w (Step 2.c.vii). ☐

An immediate corollary of the above theorem is that all observable behaviors of M_V are generated by the algorithm of Figure 1, which can thus be used to check whether $L_O(M_V) \subseteq L(P)$.

We can also prove that the optimization for constraint nodes performed by Step 2.c of the algorithm of Figure 2 is *optimal* in the following sense.

THEOREM 3.5. *Let $M_V = (I, S, s_0, T_V)$ and $M' = (I, S, s_0, T')$ be defined as in Theorem 3.4. Let s be any reachable state s in M_V such that $n = node(s)$ and n is a constraint node. For any given set $\{v_a | a \in E\}$ of data values associated to input events enabled when the node is entered, if $n \xrightarrow{w} n'$ in M_V with $n \neq n'$, then there exists* exactly one *transition $n \xrightarrow{w} n'$ in M'.*

PROOF. (Sketch) By contradiction. Assume that there exists two transitions $n \xrightarrow{w} n'$ in M'. This implies that there exist two predicates ϕ_i and ϕ_j of n that are both violated by the set $\{v_a | a \in E\}$ of data values, whose signatures are not included in each other, and such that $target(\phi_i) = target(\phi_j) = n'$. Since both transitions $n \xrightarrow{w} n'$ are labeled by w, we have $(\sigma(\phi_i) \cap O) = (\sigma(\phi_j) \cap O) = \{a | a \in w\}$. Therefore, by the condition of Step 2.c.iii.A of the algorithm of Figure 2, ϕ_i and ϕ_j cannot be both be added to the set $Marked$, and hence the algorithm cannot visit node n' twice by executing from s two different sequences of events w' and w'' such that $w'_{|O} = w''_{|O} = w$. ☐

4. IMPLEMENTATION ISSUES

To automatically and systematically test Sisl programs for violation of safety properties using the algorithms presented in the previous section, we have integrated VeriSoft and Sisl.

VeriSoft is a tool for systematically exploring the state spaces of systems composed of several concurrent processes executing arbitrary code written in any language. The state space of a system is a directed graph that represents the combined behavior of all the components of the system. Paths in this graph correspond to sequences of operations (scenarios) that can be observed during executions of the system. VeriSoft systematically explores the state space of a system by controlling and observing the execution of all the components, and by reinitializing their executions. VeriSoft drives the execution of the whole system by intercepting, suspending and resuming the execution of specific operations (system calls) executed by the implementation being tested. Examples of operations intercepted by VeriSoft are operations on communication objects (e.g., sending or receiving a message), and the VS_toss(n) operation mentioned earlier, which simulates nondeterminism and introduces a branching point with $n + 1$ branches in the state space whenever it is executed. VeriSoft can always guarantee a complete coverage of the state space up to some depth; in other words, *all* possible executions of the system up to that depth are guaranteed to be covered. Since VeriSoft can typically generate, execute and evaluate thousands of tests per minute, it can quickly reveal behaviors that are virtually impossible to detect using conventional testing techniques. More details about the state-space exploration techniques used by VeriSoft are given in [11]. VeriSoft has been applied successfully for analyzing several software products developed in Lucent Technologies, such as telephone-switching applications and implementations of network protocols (e.g., see [12]).

In order to use VeriSoft for controlling the execution of the nondeterministic algorithms from Section 3, we have built a "VeriSoft interface" to Sisl. This interface provides the necessary information requested by the algorithms of the previous section (such as the current set of enabled input events, etc.). These algorithms were implemented in a straightforward manner in Java. Calls to the external operation VS_toss were performed using the Java Native Interface. VeriSoft can then control the execution of the resulting single process formed by the combination of the Sisl application being tested and its nondeterministic test driver, by intercepting calls to VS_toss and providing the value returned by these calls, and by creating and destroying the Java Virtual Machine to reinitialize the program.

For testing of safety properties, we used the specification-based testing package of Triveni [6], a framework for event-driven concurrent programming in Java. This implementation uses a standard algorithm [20] to translate a given safety formula in propositional linear-time temporal logic formulas into a finite-state automaton whose language is the set of finite event sequences that violate the formula.

5. EXAMPLE AND EXPERIMENTS

Consider Table 1 which describes an interactive banking service called the Teller.

To motivate the Sisl implementation of this service, we describe the transfer of funds in more detail. The transfer capability establishes a number of constraints among the three input events (source account, target account, and transfer amount) required to make a transfer:

- the specified source and target accounts both must be valid accounts for the previously given (login,PIN) pair;
- the dollar amount must be greater than zero and less than or equal to the balance of the source account;
- it must be possible to transfer money between the source and target accounts.

These constraints capture the minimum requirements on the input for a transfer transaction to proceed. Perhaps more important is what these constraints do not specify: for example, they do not specify an ordering on the three inputs, or what to do if a user has repeatedly entered incorrect information.

Figure 3 depicts the Sisl service logic for the Teller, specified as a reactive constraint graph. It uses constraint nodes to describe the requirements on the transfer capability, deposit capability, and withdrawal capability. In particular, there is a constraint node for each transaction type. In order to enter the service, the user must first provide a *startService* event (e.g. dialing into the service or going to the web page); this is not depicted in Figure 3. The user then must successfully log in with a valid login and pin combination. Since the login and pin may be provided in either order, a constraint node expresses this requirement. For expository simplicity, we assume that the login and pin must be identical for the login to be successful. After the user has successfully logged in, the service provides a choice among the different transaction types. If a *startTransfer* event is provided, for example, control flows to the transfer constraint node. The user is then prompted for a source account, target account and amount, in any order. If the user provides a source account and an amount which is greater than the balance of the source account, for example, the constraint `amt <= Balance(src)` will be violated. Since no explicit failure target nodes have been specified, control flow will remain in the current node. If the user provides consistent information about both accounts and the amount, the transfer will be performed and control reverts back to the choice node on transaction types. The self-loop annotated with "!has_Quit" on the choice node indicates that the the subgraph from this node will be repeatedly executed until the precondition becomes false (i.e. the user has quit the service).

Some temporal properties of interest for the Teller include:

- The service can accept a source account only if the current transaction type is either a withdrawal or transfer.
- The service can accept a target account only if the current transaction type is either a deposit or transfer.
- The service can begin a deposit transaction only if the user has given a login and pin in the past and has not quit the service.

Each property described above in English has an equivalent formula in propositional linear-time temporal logic. In our terminology, the set of observable events when analyzing each property is the set of events that occur in the corresponding formula. For example, the set of observable events when analying the first property is {src,startWithdrawal, startDeposit, startTransfer}; in particular, the reference to the "last" transaction type necessitates the *startDeposit* event to occur in the corresponding formula.

In our implementation, the two valid accounts are checking and savings, and transfers are permitted only between accounts of different types; i.e. between checking and savings, and vice-versa. Money market accounts are not considered to be valid accounts in this example. Initially, the balance on all accounts is zero, and the hasQuit variable is set to false.

Our implementation of this portion of the Teller consists of approximately 200 lines of text in a mark-up language, which is automatically translated by the Sisl toolset into approximately 500 lines of Java code. It currently has applet, HTML, automatic speech recognition, and VoiceXML [21] interfaces, each about 300 lines, all sharing the same service logic.

5.1 Results of Experiments

To evaluate our approach and compare the efficiency of the algorithms presented in Section 3, we performed systematic state-space explorations on the Teller service.

The Teller is an interactive banking service. The service is login protected; the customer must authenticate themselves by entering an identifier (login) and PIN (password) to access the functions. As customers may have many money accounts, most functions require the customer to select the account(s) involved. Once authenticated, the customer may:

Make a deposit.

Make a withdrawal. The service makes sure the customer has enough money in the account, then withdraws the specified amount.

Transfer funds between accounts. The service prompts the customer to select a source and target account, and a transfer amount, and performs the transfer if (1) the customer has enough money in the source account, and (2) transfers are permitted between the two accounts.

Quit the service.

Table 1: A high-level description of the Teller.

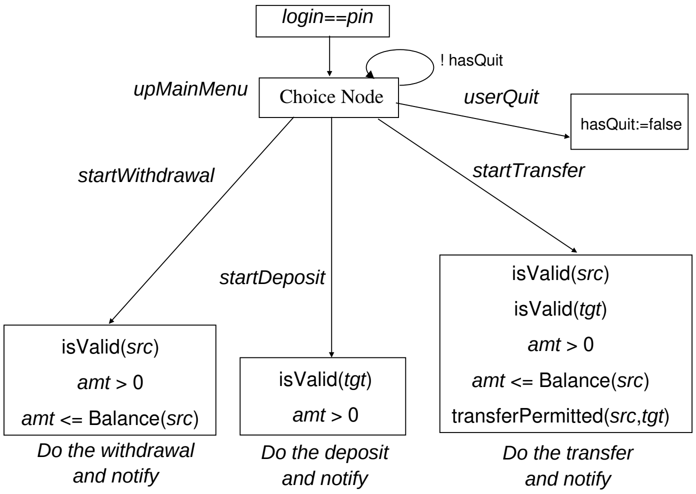

Figure 3: The Reactive Constraint Graph for the Teller

We first selected the following data to be associated to each event: the *name* and *pin* events each were assigned two names from the same set {John, Mary}, the *src, tgt, amt* events each were assigned three types from the set {checking, savings, moneymarket}, and the *amt* event was assigned values from the set $\{0, 100\}$.

For the analysis, we first tested the following temporal property: the service can accept a target account only if the current transaction type is a deposit. The set of observable events of this property is {tgt, startDeposit, startWithdrawal, startTransfer}. As expected, both algorithms reported a violation trace in which the current transaction type is a transfer and a target account was accepted.

We then ran a set of experiments in which the set of observable events was empty, in order to measure the efficiency of the algorithms. This actually tests that there were no uncaught run-time exceptions in the Sisl (Java) program along all paths up to a certain depth, as measured in the number of events sent to the service logic. We ran these tests in succession, each time increasing the maximum depth of the paths to be explored. Our results are summarized in the Figure 4. The plot on the left depicts the number of paths explored by the algorithms against the maximum depth of paths to be explored, while the plot on the right depicts the running time of the algorithms in seconds against the maximum depth of paths to be explored.

Some interesting observations can be made about the experimental data. First, as expected, the running times of both algorithms are proportional to the number of paths explored. Second, consider the rate at which the number of paths and running times increase with respect to the maximum path depth; this rate is significantly less for the optimized algorithm.

An important observation about the experimental data is that for maximum depths of 5 and 6, the optimized algorithm explores the same number of paths and hence has the same running time! This phenomenon occurs because the optimized algorithm performs the bulk of its work upon entry into a constraint node. This is especially true in the case of an empty set of observable events: in this case, all the work is done by the optimized algorithm upon entry into the constraint node. For example, in the Teller, paths of depth 4 consist of a *startService* event followed by a consistent *name* and *pin* event in either order, followed by a transaction type. At depth 5, control enters one of the transaction constraint nodes. The optimized algorithm performs all its work in choosing the event data, computing the marked predicates, and choosing a marked predicate upon entry into the node. It then merely sends the corresponding (hidden) events in a fixed order. Hence the set of paths is explored is identical at depths 5 and 6, and the running time is the same (except for the extremely minor activity of sending an additional event). The similar phenomenon occurs at depths 8, 9, and 10.

The results show that even for small examples such as the Teller, the optimized algorithm can provide a significant improvement in efficiency: for example, at depth 11, the simple algorithm takes over one and a half hours to complete, while the optimized algorithm takes only eleven minutes. Hence, the latter can be used to efficiently and systematically test much more complicated services.

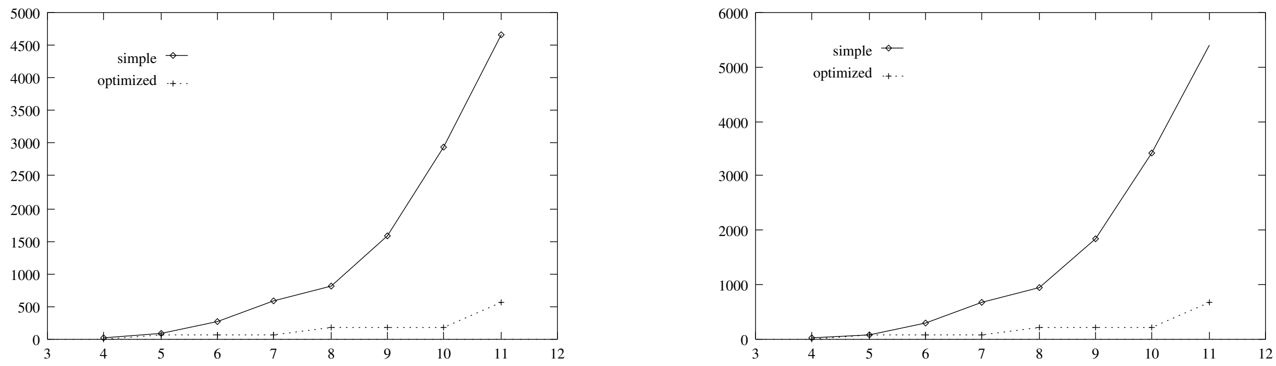

Figure 4: Experimental Results for Number of Paths Explored (left) and Running Time in Seconds (right) vs. Path Depth

6. CONCLUDING REMARKS

6.1 Sisl Applications

We are currently using Sisl in several projects involving multi-modal interactive services. For example, Sisl is being used to prototype a new generation of call processing services for a Lucent Technologies switching product. As part of this research/development collaboration, we are developing some call processing features that may form the basis of new product offerings. The service logics for these features are expected to be quite complex, and need to be tested thoroughly. We are planning to use the techniques and tools presented in this paper to test these applications.

We are also planning to test two other Sisl applications being developed at Lucent Technologies: an interactive service based on a system for visual exploration and analysis of data, and some collaborative applications in which users may interact with the system through a rich collection of devices.

6.2 Related Work

We conclude with a comparison of our approach with other related work.

Combining an open reactive system with its most general environment is related to the idea of "hiding" a set of visible actions of a process in a process calculus [13, 17].

Closing automatically open reactive (event-driven) programs for systematic testing (model-checking) purposes has been studied in [5, 8]. For sequential (data-driven) programs, numerous algorithms have also been proposed to automatically generate a set of input data that is sufficient to exercise and test all the possible paths in the control-flow graph of a program, for instance. This previous work makes extensive use of static analysis techniques (e.g., [7, 18, 2]), which automatically extract information about the dynamic behavior of a sequential program by examining its text. In contrast, the algorithms presented here dynamically detects at run-time the set of input events that the application under test is currently ready to accept, and uses that information to drive its execution, without using any static analysis techniques. This makes our algorithms directly applicable to any host language (Java, Perl scripts, etc.) and environment (including external databases, etc.).

The observation exploited by our second algorithm, namely that interleavings of input events at a constrained node have sometimes the same effect on the overall behavior of the system, is somewhat similar to the intuition behind partial-order reduction algorithms used in model-checking to prune the state spaces of concurrent systems (e.g., see [10]). A major difference is that these algorithms exploit a notion of "independence" (commutativity) on actions executed by interacting concurrent processes. In contrast, constraint-based programs are purely sequential. The reduction we obtain here is derived directly from the structure of the program and takes advantage of a form of symmetry induced by constraint nodes and their ability to specify *sets* of input events rather than single events. Another example of programming language construct inducing symmetry that can be exploited during verification (systematic testing) is the "scalarset" [14].

7. REFERENCES

[1] Abella, A., Brown, M., and Buntschuh, B. (1996). Development principles for dialog-based interfaces. In *Proceedings of the European Conference on Artificial Intelligence*, volume W29, pages 1–6, Budapest, Hungary. Wiley.

[2] Aho, A., Sethi, R., and Ullman, J. (1986). *Compilers: Principles, Techniques and Tools*. Addison-Wesley.

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

[4] Ball, T., Colby, C., Danielsen, P., Jagadeesan, L., Jagadeesan, R., Läufer, K., Mataga, P., and Rehor, K. (1999). Sisl: Several interfaces, single logic. *International Journal of Speech Technology*. Accepted for publication.

[5] Colby, C., Godefroid, P., and Jagadeesan, L. J. (1998a). Automatically Closing Open Reactive Programs. In *Proceedings of 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 345–357, Montreal. ACM Press.

[6] Colby, C., Jagadeesan, L. J., Jagadeesan, R., Läufer, K., and Puchol, C. (1998b). Design and implementation of Triveni: a process-algebraic API for threads + events. In *Proceedings of the International Conference on Computer Languages*, pages 58–67, Chicago, IL. IEEE Computer Society Press.

[7] Cousot, P. and Cousot, R. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*.

[8] Dwyer, M. B. and Pasareanu, C. S. (1998). Filter-based Model Checking of Partial System. In *Proceedings of SIGSOFT'98 Sixth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 189–202.

[9] Goddeau, D., Meng, H., Polifroni, J., Seneff, S., and Busayapongchai, S. (1996). A form-based dialogue manager for spoken language applications. In *Proceedings of the International Conference on Spoken Language Processing*, volume 2, pages 701–704, Philadelphia, PA. IEEE.

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

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

[12] Godefroid, P., Hanmer, R. S., and Jagadeesan, L. J. (1998). Model Checking Without a Model: An Analysis of the Heart-Beat Monitor of a Telephone Switch using VeriSoft. In *Proceedings of ACM SIGSOFT ISSTA'98 (International Symposium on Software Testing and Analysis)*, pages 124–133, Clearwater Beach.

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

[14] Ip, C. N. and Dill, D. L. (1993). Better verification through symmetry. In Agnew, D., Claesen, L., and Camposano, R., editors, *Proceedings of the 1993 Conference on Computer Hardware Description Languages and their Applications*.

[15] Issar, S. (1997). A speech interface for forms on WWW. In *Proceedings of the European Conference on Speech Communication and Technology*, pages 1343–1346, Rhodes, Greece. European Speech Communication Association.

[16] Jaffar, J. and Lassez, J. L. (1987). Constraint logic programming. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages*.

[17] Milner, R. (1989). *Communication and Concurrency*. Prentice Hall.

[18] Muchnick, S. and Jones, N. (1981). *Program Flow Analysis: Theory and Applications*. Prentice-Hall.

[19] Saraswat, V. A. (1993). *Concurrent Constraint Programming*. Logic Programming and Doctoral Dissertation Award Series. MIT Press.

[20] Vardi, M. and Wolper, P. (1986). An automata–theoretic approach to automatic program verification. In *Proc. LICS*, pages 332–339.

[21] VoiceXML (1999). http://www.voicexml.org/.

[22] Wang, K. (1998). An event driven model for dialogue systems. In *Proceedings of the International Conference on Spoken Language Processing*, volume 2, pages 393–396, Sydney, Australia. Australian Speech Science and Technology Association, Incorporated.