Sisl: Several Interfaces, Single Logic

Thomas Ball[‡*] Christopher Colby[+] Peter Danielsen[†] Lalita Jategaonkar Jagadeesan[†]
Radha Jagadeesan[+] Konstantin Läufer[+] Peter Mataga[†] Kenneth Rehor[†]

[‡] Microsoft Research
One Microsoft Way
Redmond, WA 98052
tball@microsoft.com

[†]Software Production Research Dept.
Bell Laboratories, Lucent Technologies
263 Shuman Blvd.
Naperville, IL 60566
{wiscal,lalita,mataga,krehor}@research.bell-labs.com

[+]Dept. of Mathematical and Computer Sciences
Loyola University Chicago
6525 N. Sheridan Road Chicago, IL 60626
{colby,radha,laufer}@cs.luc.edu

January 6, 2000

ABSTRACT

Modern interactive services such as information and e-commerce services are becoming increasingly more flexible in the types of user interfaces they support. These interfaces incorporate automatic speech recognition and natural language understanding, and include graphical user interfaces on the desktop and web-based interfaces using applets and HTML forms. To what extent can the user interface software be decoupled from the service logic software (the code that defines the essential function of a service)? Decoupling of user interface from service logic directly impacts the flexibility of services, or, how easy they are to modify and extend.

To explore these issues, we have developed Sisl, an architecture and domain-specific language for designing and implementing interactive services with multiple user interfaces. A key principle underlying Sisl is that all user interfaces to a service share the same service logic. Sisl provides a clean separation between the service logic and the

[*] This work was conducted while the author was at Bell Laboratories, Lucent Technologies.

software for a variety of interfaces, including Java applets, HTML pages, speech-based natural language dialogue, and telephone-based voice access. Sisl uses an event-based model of services that allows service providers to support interchangeable user interfaces (or add new ones) to a single consistent source of service logic and data.

As part of a collaboration between research and development, Sisl is being used to prototype a new generation of call processing services for a Lucent Technologies switching product.

KEYWORDS

interactive services, domain-specific languages, web, automatic speech recognition, dialogue systems, telephony, voice services, user interfaces, Triveni, Java, VoiceXML

1 INTRODUCTION

Modern interactive services are becoming increasingly more flexible in the user interfaces they support. These interfaces incorporate automatic speech recognition (ASR) and natural language understanding, and include graphical user interfaces on the desktop and web-based interfaces using applets and HTML forms. Such services include banking systems, airline reservation systems, stock market services, and Internet call centers, to name but a few.

A basic problem in the development of such systems is that the expense associated with the user interfaces can be significant. For example, approximately 50% percent of the software for systems with highly-interactive graphical interfaces is dedicated to the user interface [Myers and Rosson, 1992]. This difficulty is accentuated by the wide variety of possible interfaces to modern interactive services. In general, the information presented on each device must be tailored to the functionality that is available on that device. Thus, the graphical interface to an application that is appropriate for a workstation may be quite different than an interface that is appropriate for an interactive voice response system or a cellular phone with a very small display. Last but not least, natural language dialogue interfaces based on automatic speech recognition add new flexibility in interaction with the user.

To what extent can the user interface software be decoupled from the service logic software (the code that defines the essential function of a service)? In general, how does one achieve the following modularity principles?

- Reduce the amount of changes to the user interface software required due to modifications to the service logic software, and vice-versa;
- Enable service providers to provide *interchangeable* user interfaces (or add new ones) to a single consistent source of service logic and data.

A first step in this direction is a software architecture following standard ideas from component or distributed programming architectures. Such an architecture specifies the connections and communication between the service logic and user interfaces as abstract interfaces, and permits addition of new user interfaces as long as they comply with the interface specifications. Thus, such an architecture allows several different user interfaces to share the same service logic programming code.

However, a general software architecture does not address issues regarding succinct expression of the service logic. Let us say that we want to add a new user interface to an existing service logic. The architecture viewpoint certainly allows this as long as the code for the new user interface satisfies the interface requirements. However, there is no guarantee that the "old" service logic permits the "new" user interface to be used in an unfettered and completely flexible fashion. This issue is especially significant because of the fundamental differences in the nature of interaction across the spectrum of user interfaces. For example, in typical web interaction based on HTML forms, the buttons and

The Any-Time 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.
Get the balance of an account. Display the balance with respect to all posted transactions.
View the transactions of an account. Display the transactions for the selected account.
See the last transaction. Display the last action the customer performed.

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

fields of the form serve to constrain the type of information that the user can provide to the service logic. In interactive voice response (IVR) systems, such restrictions are imposed by listing out menus to users (“ for ..., press 1, for .., press 2” etc) and forcing the user to stay within the listed options. However, when users interact with the system through spoken natural language, control is shared between the user and the system. In a free flowing conversation, the system can take control by requesting required information from the user, and can verify, correct, or clarify information given by the user. Importantly, however, the user can also query or direct the system at any time by providing information that has not been requested by the system. Consequently, the fact that a service logic is adequate for user interaction based on HTML forms and IVR does not guarantee that it will be adequate for a user interface based on automatic speech recognition, since it does not necessarily support the additional flexibility of the part of the user. To allow this, the service logic has to be designed to support multiple user interfaces by providing a high-level abstraction of the service/user interaction.

We motivate our approach through an example. Consider Table 1, which describes an interactive banking service called the Any-Time Teller (a running example in this paper). 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. Thus, the basic principle of designing a flexible service logic might be restated:

> *Introduce a constraint among events only when absolutely necessary for the correct functioning of a service.*

This basic principle leads to a number of sub-principles of service specification that directly address the issues of flexibility and decoupling. For example, the service logic should be able to:

- accept input in different orders;
- accept incomplete input;
- allow the user to correct or update previously submitted information;
- automatically send to the user interfaces information about user prompts, help, and ways to revert back to previous points in the service.

There are two basic requirements on user interfaces that can be used in conjunction with such service logics. In particular, any such user interface must be able to:

- Based on information received from the service logic, prompt the user to provide the appropriate information and respond if the user needs help;
- Collect information from the user and transform the information into abstract events to be sent to the service logic.

A wide range of user interfaces satisfy these requirements. For example, web-based interfaces using HTML pages can perform this event transformation and communication via name/value pairs, as can telephone-based voice interfaces based on VoiceXML [VoiceXML, 1999] document interpretation. Desktop automatic speech recognition interfaces based on the Java Speech API support this capability via tags in the speech grammars (the input to a speech recognition engine that permits it to efficiently and effectively recognize spoken input).

Based on the above principles, we have developed Sisl (Several Interfaces, Single Logic), an approach for designing and implementing interactive services with multiple user interfaces. A key idea underlying Sisl is that all user interfaces to a service share the same service logic, which provides a high-level abstraction of the service/user interaction. It thus modularizes the development process by allowing the implementation of the service logic and user interfaces to proceed independently.

There are two pieces to the Sisl approach: (1) a standard language independent architecture and (2) *reactive constraint graphs*, a novel domain-specific language (DSL) for designing and implementing interactive services. Programs written in the DSL are used as components in the architecture. The DSL is based on an analysis of the features required of a service logic that is shared across many different user interfaces, including speech-based natural language interfaces. Among other things, our DSL permits the description of service logics that can accept partial and incomplete information from the user interface, and allows user interfaces to deliver events to the service logic in any order.

Sisl is implemented as a library on top of Triveni [Colby et al., 1998a, Colby et al., 1998b], a process algebraic API for reactive programming. Triveni itself has been implemented as a library in Java, enabling Sisl to be used with any (Personal) Java implementation. Sisl has a XML front-end to describe reactive constraint graphs in a markup language, allowing Sisl to be used easily by application programmers. The Sisl implementation currently supports applet-based interfaces, HTML interfaces using the Java Servlet API, speech-based natural language interfaces using the Java Speech API, and telephone-based voice access via VoiceXML [VoiceXML, 1999] and the TelePortal platform [Atkins et al., 1997]. Our implementation of the Sisl language consists of approximately 2300 lines of Java and Triveni code (in addition to the Triveni library, which is about 6000 lines of Java code). The implementation of the infrastructure for the supported interfaces described above is about 500 lines each.

The rest of this paper. In Section 2, we present the criteria for designing and implementing service logics that are shared by several interfaces, and discuss how current approaches perform with respect to these criteria. In Section 3, we describe the top level architecture of Sisl, including the protocols on the communication between service logic and its multiple user interfaces. The requirements on the implementations of user interfaces are described in Section 4, while Section 5 presents the language of reactive constraint graphs. In Section 6, we illustrate the concepts with the implementations and execution traces of part of the functionality of the Any-Time Teller service. This service has an applet, web, automatic speech recognition, and telephone-based voice interface, all sharing the same service logic. Section 7 describes additional features of Sisl, including its associated testing/debugging facility. In Section 8, we conclude by describing briefly some applications in Sisl, and discuss future work.

2 Criteria for Flexible Interactive Service Creation

We now discuss the principles for flexible interactive services that guided the design of Sisl, and describe how current approaches perform with respect to these criteria.

2.1 Principles for a Domain-Specific Language

As described in the introduction, a basic principle of flexible service logics is that *the service logic should introduce a constraint among events only when absolutely necessary for the correct functioning of the service.* We now elaborate on this idea and its consequences, using speech and web-based interfaces to illustrate the concepts.

Partial order on events. The service should implement a partial order on events, rather than a total order on events (in the spirit of [Hayes et al., 1985]).

For example, consider an implementation of the transfer capability that totally orders the interactions of the service with the user interface: namely, the source account is first collected and checked for validity, then the amount is collected and another validity check is performed, and finally, the target account is collected and other validity checks are performed.

Such an implementation suffices in the context of a web based HTML interface which serves up forms in a *fixed* order and thus controls the order of receipt of information from the user interface.

However, such an implementation overly restricts a speech interface, since robust speech interfaces should allow the information to be collected in any order. In particular, the user should be allowed to say

- “I want to transfer $500 from my savings to my checking account” or
- “I want to transfer $500 to my checking account from my savings account” or
- “From my savings account, transfer $500 to checking”

Thus, the source account, amount, and target account can be given in any order.

By making explicit independent and dependent events, it becomes clear what events may be reordered without affecting the behavior of the service.

Partial information. The service should be able to accept partial information in the form of “incomplete” input. Consider the following speech recognition scenario for fund transfer:

- User says “I want to transfer $500 dollars from my savings account.”
- The user interface, after querying the service logic, prompts the user for the target account.

- The user then specifies the target account.

An implementation of the transfer capability that reacts only when all three events (source account, target account, and amount) are presented checks that the savings account has $500 dollars only after the last step. In the scenario where the user's savings account does not contain $500, the user will have gone through several needless interactions before this is reported by the system.

Since a constraint on the input events may refer to any arbitrary subset of these events, it is desirable that the service logic should be able to accept arbitrary subsets of events at any time.

Error recovery. Unfortunately, humans often change their mind and/or make mistakes. Whenever possible, services must accommodate these shortcomings of our species, providing a capability to correct or update previously submitted information, back out of a transaction and allow the user to back up to previous points in the service.

User prompts. The service logic must automatically send to the user interfaces information about user prompts, help, and ways to revert back to previous points in the service.

This principle is particularly relevant for services that obey the previously stated three principles. Such services allow multiple points of interaction to be potentially enabled at a given instant to accommodate the different orderings of inputs. In such a context, the prompt and help information serves as an abstraction of the current control point of the service, and can be handled in a different manner by different user interfaces. In automatic speech recognition interfaces, the information about currently enabled events is used by the user interface in two ways [JSAPI, 1998]:

- to appropriately prompt the user for information, thus compensating for the lack of visual cues, and
- to effectively parse the information provided by the user.

A user interface need not respond to all currently enabled events of the service. Thus, different user interfaces can formulate different queries to the user even though the state of the underlying service logic (as revealed by the current set of enabled events) is the same.

2.2 Prior Approaches

The traditional approach for designing and implementing interactive services is based on finite state machines (FSMs). These state machines may be described explicitly in a graphical notation such ObjecTime or StateCharts, or described implicitly in imperative languages such as Perl, C, Mawl [Atkins et al., 1999, Atkins et al., 1997] or Active Server Pages [Francis et al., 1998]. Figure 1 presents a simplified FSM that one might construct to handle the collection of the three inputs needed for the transfer transaction. Each state is uniquely identified by the set of events that has been collected so far, and each state checks the constraints for which all relevant events have been collected. By construction, the FSM is insensitive to the order in which it receives events, since it explicitly specifies every possible sequence of three events. However, we have left many arcs out of this FSM, including arcs needed to handle the violation of constraints, or the user correcting previously submitted information. This example illustrates the two major problems faced by FSM-based approaches:

- As the number of input events and constraints to be checked increases, there is the potential for an exponential blow-up in the number of states. This is especially true if the FSM has been designed to handle different orderings of input events, incomplete inputs, and correction of previous inputs.

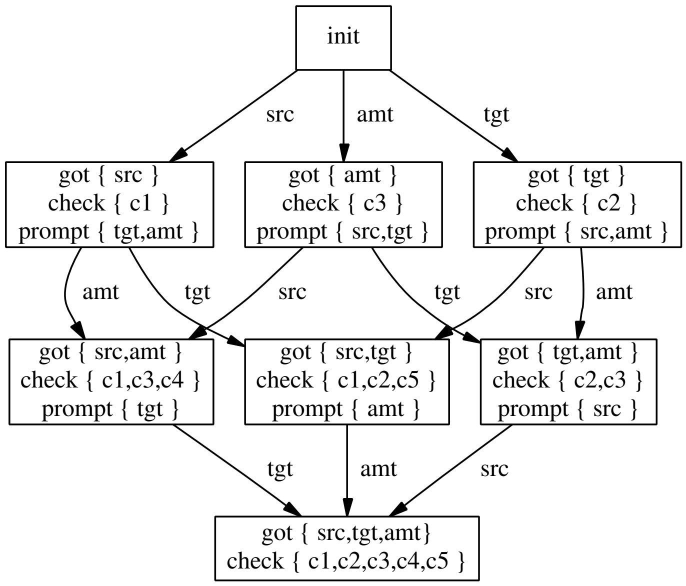

Figure 1: A (simplified) finite state machine for collecting the source account (src), target account (tgt), and dollar amount (amt) needed for the transfer transaction.

- The FSM is fragile with respect to changes in the requirements of the service. For example, if a fourth event were required for a transfer to take place, the changes to the FSM would be non-modular and hence complicated.

Reactive constraint graphs in Sisl allow explicit description of constraints, satisfy the requirements that we have described and do not suffer from these two problems of FSM-based approaches.

Similar issues also arise in spoken dialogue systems, which combine automatic speech recognition with natural language understanding.

Toolkits and application programming interfaces for building spoken dialogue systems include CSLU [Sutton et al., 1996], VIL [Pargellis et al., 1998], and REWARD [Brondsted et al., 1998]. In spoken dialogue systems, dialogue flow is controlled by a dialogue manager, which is responsible for eliciting sufficient information from the user, communicating this information to the application, and reporting information back to the user. In mixed-initiative dialogue systems, dialogue control is shared between the user and the system. Approaches to dialogue management can be broadly classified into FSM-based methods, on the one hand, and self-organizing or locally-managed approaches on the other [McTear, 1998].

- FSM-based methods face the obstacles described above, such as inflexibility of the dialogue, difficulty handling extra information provided by the user, difficulty allowing the user to back up to previous points in the service, and combinatorial explosion of the state machine size when adding more flexibility in user input [McTear, 1998, Goddeau et al., 1996].

- The self-organizing approach to dialogue management includes methods based on frames [Abella et al., 1996] and forms [Goddeau et al., 1996, Issar, 1997]; the focus is on systems with a single, spoken natural language interface. In contrast, our approach emphasizes services with multiple and varied interfaces.

Sisl extends event-based approaches aimed at multi-modal systems where dialogue control is represented as a variant of AND/OR trees [Wang, 1998] in two ways. We move from events to partial information on sets of events, and from

AND nodes to constraint nodes that wait for a constraint to be satisfied. We also augment the control structures with the ability to revert back to earlier points in the service, and provide automatic generation of prompts and user help.

3 THE Sisl ARCHITECTURE

The Sisl architecture is shown in Figure 2. All communication between the service logic and its multiple user interfaces[1] passes through the service monitor, and occurs via events. The service monitor mediates communication between the service logic and the user interfaces, on an application-specific basis. In particular, the service monitor is responsible for passing events from the user interfaces to the service logic, in a priority order that is tailored to the application.

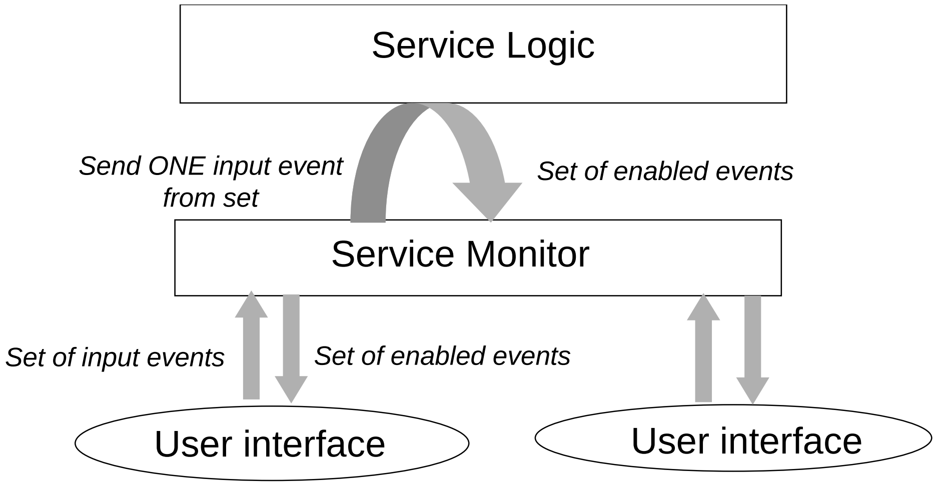

Figure 2: The Sisl Architecture

The user interface prompts and collects events from the user and dispatches these to the service monitor. The service monitor interacts with the service logic in rounds. In each round, an individual event is selected from the input set in priority order and sent to the service logic, which processes the event and performs its associated computations, and reports its new enabled set of events to the service monitor. If events in the input set match the enabled set of events, the service monitor selects one in priority order and the process is repeated. If no event in the input set matches the enabled set of events, the service monitor reports the currently enabled set of events of the service logic to the user interface. Based on these enabled events, the user interface prompts the user and collects the next set of events.

There are three kinds of events: *prompt* events, *up* events, and *notify* events.

Prompt events. Prompt events indicate to the user interface what information to communicate to the user, and what information the service is ready to accept. There are three kinds of prompt events:

- *prompt_choice* events are disjunctive choices currently enabled in the service logic. For example, after the user has successfully logged into the Any-Time Teller, a choice among the different transaction types is enabled. The service logic sends a *prompt_choice_deposit*, *prompt_choice_withdrawal*, *prompt_choice_transfer* event, and so forth, to the service monitor.

[1] There is one instance of the service logic for each user interacting with the service.

- *prompt_req* events are the events currently required by the service logic. For example, suppose the user has chosen to perform a transfer transaction. The Any-Time Teller requires that the user input a source account, target account, and amount, and hence sends a *prompt_req_src*, *prompt_req_tgt*, and *prompt_req_amt* event to the service monitor.

- *prompt_opt* events are events enabled in the service logic for which the user may correct previously given information. For example, suppose the user is performing a transfer and has already provided his source and target accounts, but not the amount. The service logic sends a *prompt_opt_src*, *prompt_opt_tgt* event, as well as a *prompt_req_amt* event, to the service monitor. This indicates that the user may override the previously given source and target accounts with new information.

Up events. Up events correspond to earlier points in the service logic to which the user may go back. For example, the service logic sends an *up_MainMenu* event to the service monitor. This allows the user to abort any transaction and go back up to the main menu.

Notify events. Notify events are simply notifications that the user interface should give the user, for example, that a transaction has completed successfully or that information provided by the user was incorrect or inconsistent.

4 REQUIREMENTS ON USER INTERFACES

User interfaces have two main responsibilities with respect to the Sisl architecture that reflect the two-way information flow between the user interface and service logic:

- Based on the events received from the service logic (via the service monitor), prompt the user to provide the appropriate information and respond if the user requests help.

- Collect the information from the user and transform the information into events to be sent to the service logic, via the service monitor.

Any user interface that performs these functions can be used in conjunction with a Sisl service logic. For example, web-based interfaces using HTML pages can perform this event transformation and communication via name/value pairs, as can telephone voice interfaces based on VoiceXML document interpretation. Desktop automatic speech recognition interfaces based on the Java Speech API support this capability via tags in the speech grammars (the input to a speech recognition engine that permits it to efficiently and effectively recognize spoken input).

In addition, Sisl provides a convenient framework for designing and implementing web, applet, automatic speech recognition, and telephone-based voice interfaces. To implement such user interfaces, the UI designer need only specify two functions – corresponding to the prompt and help mechanisms. For automatic speech recognition interfaces (on the desktop and via the telephone), a set of speech grammars together with a third function that specifies which grammars to enable is also required.

The required functions are:

- A *prompt* function that generates the string to be given as the prompt to the user. An example is given in Table 2. The Sisl infrastructure automatically causes speech-based interfaces to speak the prompt string. Web-based interfaces automatically display the prompt string, as well as radio buttons corresponding to the possible transaction choices. (For the other prompt events, text fields are automatically displayed, while submit buttons are automatically displayed for enabled *up* events.) A screen snapshot is given in Figure 3.

```
prompt(req_events, opt_events, choice_events, uplabels) {
    ...
    if (req_events.contains_any_of("startDeposit", "startWithdrawal",
                                      "startTransfer","startBalance") {
        return("What transaction would you like to do?");
    };
    ....

    if (req_events.contains("startTransfer"))
        transaction_type.set("Transfer");
    ...

    if (transaction_type.equals("Transfer")) {
        if (req_events.contains({"src", "tgt", "amount"})) {
            return("Please specify the source account, target account,
                    and the amount you would like to transfer.")
        }
    .....
    };
}

help(req_events, opt_events, choice_events, uplabels) {
     ....
    if (req_events.contains_any_of("startDeposit", "startWithdrawal",
                                      "startTransfer","startBalance") {
        return("You may make a deposit, withdrawal or transfer.  Or you may quit the service");
    }
}
```

Table 2: A portion of the Web, Desktop ASR and Telephone-based Voice User Interfaces for the Any-Time Teller

- A *help* function that generates the string to be given as the prompt to the user. An example is given in Table 2.
- A *grammar* function that enables the correct set of grammar rules. This function is only needed for automatic speech recognition interfaces. An example is given in Table 4.

Table 2 shows portions of the prompt and help functions shared by a web, automatic speech recognition, and telephone-based voice interface for the Any-Time Teller. Portions of the grammar rules, against which the speech recognition engine will parse spoken input from the user, are depicted in Table 3. Table 4 illustrates a portion of the associated grammar function shared by an automatic speech recognition interface and telephone-based voice interface.

From these functions (and grammars), the Sisl infrastructure automatically coordinates the collection and event transformer mechanisms, and integrates the user interface with the service logic and the service monitor. For automatic speech recognition-based interfaces, the Sisl infrastructure automatically generates a desktop interface based on the Java Speech API. To enable telephone-based voice access to the service, Sisl automatically generates VoiceXML [VoiceXML, 1999] pages, which specify the voice dialog to be carried out on a telephony platform. For web-based interfaces, the Sisl infrastructure automatically generates HTML pages. (We note that Sisl also provides a mechanism for the UI designer to customize the look and feel of the interface, if so desired.)

5 SERVICE LOGIC: REACTIVE CONSTRAINT GRAPHS

In Sisl, the service logic of an application is specified as a *reactive constraint graph*, which is a directed acyclic graph with an enriched structure on the nodes. The traversal of reactive constraint graphs is driven by the reception of events

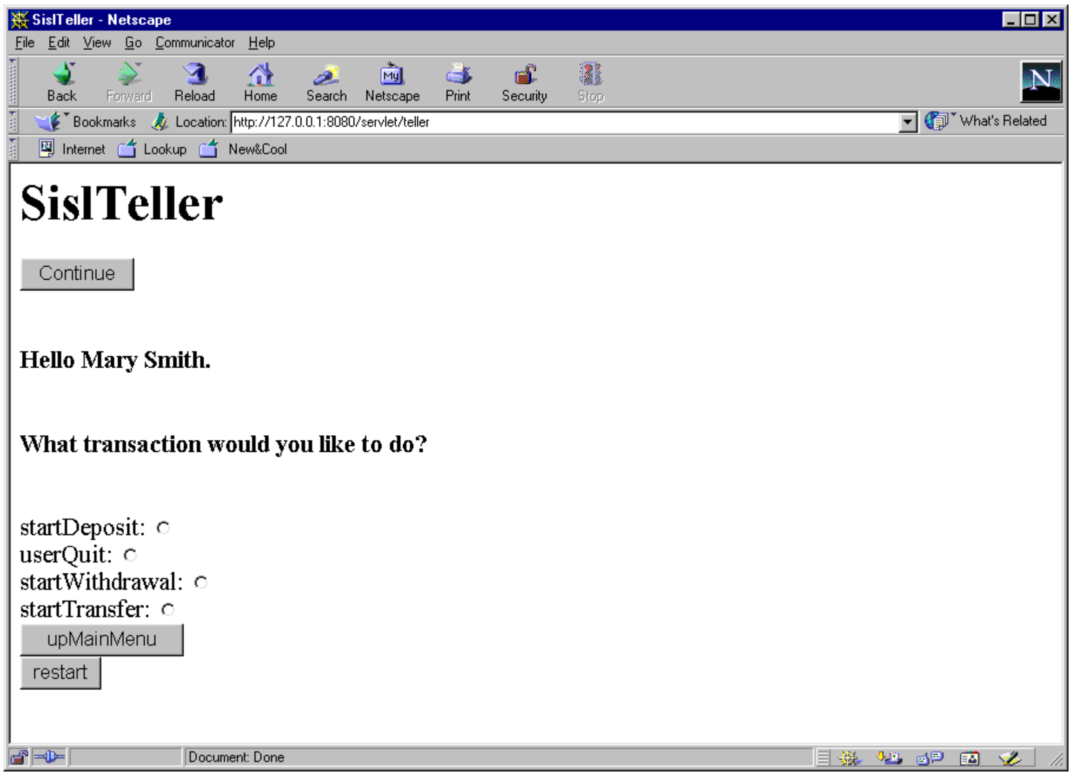

Figure 3: A Screen Shot of Web Interface: Choice Node

```
<request> = ([I (want to | would like to)] | I'd like to )| please;

<transfer_request> = [<request>] (make a transfer | transfer [money]) {startTransfer};

public <transfer> = <transfer_request> [<src_tgt_amount> | <src_tgt> | <src_amount> |
                                        <tgt_amount> | <src> | <tgt> | <amount>];

public <src_tgt_amount> = [<transfer_request>]
                          (<sum> from [my] <account_type> {src} [account]
                           (to | into)   [my] <account_type> {tgt} [account] |
                           <sum> (to   | into) [my] <account_type> {tgt} [account]
                           from [my] <account_type> {src} [account] )|
                           from [my] <account_type> {src} [account],
                           [<transfer_request>] (<sum>)
                           (to | into)   [my] <account_type> {tgt} [account];

public <src_amount> = [<transfer_request>] (<sum> from [my] <account_type> {src} [account]) |
                      from [my] <account_type> {src} [account], [<transfer_request>] (<sum>);
....
<uprequest> = [<request>] [go] [(back [up])| up] [to] [the];

public <upMainMenu> =  [<uprequest>]Main Menu {upMainMenu};
```

Table 3: A portion of the speech recognition grammar for the Desktop ASR and Telephone-based Voice User Interface for the Any-Time Teller

```
enableRules(req_events, opt_events, choice_events, uplabels) {

    evts = req_events + opt_events + choice_events + uplabels;

    if (evts.contains({"src", "tgt", "amount"})) {
        grammar.enable("<src_tgt_amount>");
    }
    if (evts.contains({"src","amount"})) {
        grammar.enable("<src_amount>");
    }
    ....
    if (uplabels.contains("upMainMenu")) {
        grammar.enable("<upMainMenu>");
    }
}
```

Table 4: A portion of the Desktop ASR and Telephone-based Voice User Interface for the Any-Time Teller

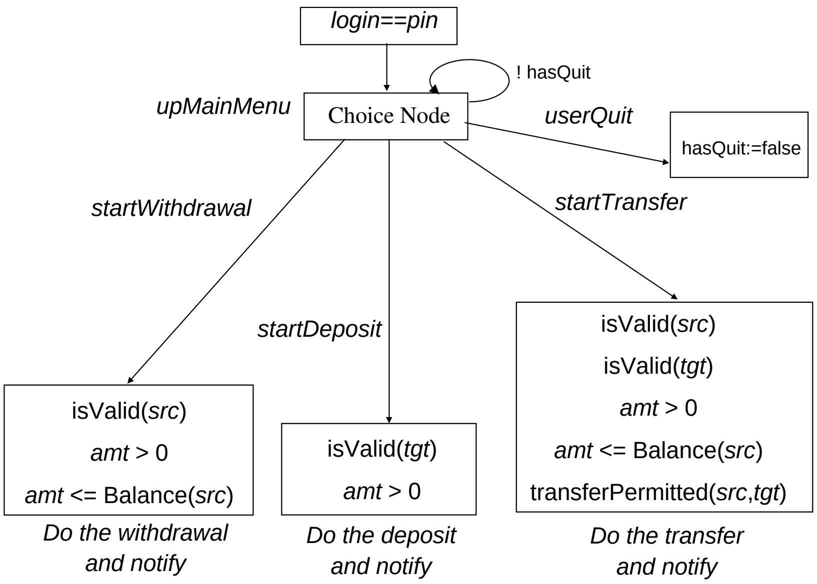

Figure 4: The Reactive Constraint Graph for a portion of the Any-Time Teller

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. An important aspect of our computational paradigm is that every reaction of the graph is atomic with respect to events: that is, no event can be received from the environment during a reaction of the graph.[2]

For example, Figure 4 depicts a Sisl reactive constraint graph that implements part of the functionality of the Any-Time Teller. As mentioned in the introduction, Sisl has a XML front-end to describe reactive constraint graphs in a markup language, allowing Sisl to be used easily by application programmers.

Types of nodes. Reactive constraint graphs can have three kinds of nodes.

Choice nodes. These nodes represent a disjunction of information to be received from the user. There are two forms of choice nodes: event-based and data-based. Every event-based choice node has a specified set of events. For every event in this set, the Sisl infrastructure automatically sends out a corresponding *prompt_choice* event from the service logic to the user interfaces (via the service monitor). The choice node waits for the user interfaces to send (via the service monitor) any 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, all outgoing transitions of a choice node must be labeled with distinct event names.

Every data-based choice node has a specified set of preconditions on data. To ensure determinism, these preconditions must be specified so that exactly one of them is true in any state of the system. When control reaches a data-based choice node, the transition associated with the unique true precondition is taken, and control transfers to the child node.

Constraint nodes. These nodes represent a conjunction of information to be received from the user. Every constraint node has an associated set of constraints on events. The Sisl infrastructure automatically sends out a *prompt_req* event – from the service logic to the user interfaces (via the service monitor) – for every event that is still needed in order to evaluate some constraint. In addition, it automatically sends out a *prompt_opt* event for all other events mentioned in the constraints. These correspond to information that can be corrected by the user.

In every round of interaction, the constraint node waits for the user interfaces to send (via the service monitor) any event that is mentioned in its associated constraints. Each constraint associated with a constraint node is evaluated as soon as all of its events have arrived. If an event is resent by the user interfaces (i.e, information is corrected), all constraints with that event in their signature are re-evaluated. For example, in Figure 4, the constraint node for the transfer capability waits for the events *src*, *tgt*, and *amt* and evaluates the corresponding constraints.

For every evaluated constraint, its optional satisfied/violated action is automatically executed, and a *notify* event is automatically sent to the user interfaces, with the specified message. If any constraint is violated, the last received event is automatically erased from all constraints, since it caused an inconsistency.

A constraint node exits when all of its constraints have been evaluated and are satisfied.

Action nodes. These nodes represent some action, not involving user interaction, to be taken. After the action is executed, control transfers to the child node.

[2]We note that reactive constraint graphs consume events. In particular, if a parent node and child node both react to a certain event, then the child node will need to wait for the next occurrence of that event: the event is said to be consumed by the parent node.

Additional structure on nodes. All nodes can have an optional "uplabel," which is used to transfer control from some descendant back up to the node, allowing the user to revert back to previous points in the service. In each round of interaction, the Sisl infrastructure automatically sends out an *up* event – from the service logic to the user interfaces (via the service monitor) – corresponding to the up-label of every ancestor of the current node.

Nodes can also have a self-looping arc, with a boolean precondition on data. This indicates that the subgraph from the node will be repeatedly executed until the precondition becomes false.

Details about constraint nodes. We now give more details about the structure of constraint nodes.

Constraints have the following components:

- The *signature* is the set of events occurring in the constraint.
- The *evaluation function* is a boolean function on the events in the signature.
- The (optional) *satisfaction tuple* consists of an optional action (not involving user interaction) and an optional notify function that may return a *notify* event with an associated message. If the constraint evaluates to true, the action is executed, the notify function is executed, and the returned *notify* event is sent to the user interfaces (via the service monitor).
- The (optional) *violation tuple* consists of an optional action (not involving user interaction), an optional notify function that may return a *notify* event with an associated message, an optional up-label function that may return the up-label of an ancestor node, and an optional child node. If the constraint evaluates to false, the action is executed, the notify function is executed, and the returned *notify* event is sent to the user interfaces (via the service monitor). The up-label function is also executed: if it returns an ancestor's up-label, it is generated, and hence control reverts back to that ancestor. If no ancestor's up-label is returned and a child node is specified, control is transferred to that node.

Every constraint node has the following components:

- An associated set of constraints. In the current semantics and implementation, this set is totally ordered[3], specifying the priority order in which constraints are evaluated.
- An optional entry action (not involving user interaction).
- An optional finished tuple, consisting of an optional exit action (not involving user interaction), an optional notify function, an optional up-label function, and an optional child node.

A detailed description of constraint node execution is given in Table 5, and summarized below.

The constraints are evaluated in the specified priority order (currently the total ordered set). If some constraint is violated and no control flow changes (via an up-label function or child node) occur as a result, the node goes back to waiting for events. If no constraints have been violated, the action and notify functions of every satisfied constraint are executed, and the returned *notify* events are sent to the user interfaces (via the service monitor). When all the constraints have been evaluated and are satisfied, the exit action and notify function associated with the constraint node are executed, and control is transferred either to an ancestor (via the up-label function) or to the child node.

[3]This may be generalized in the future to support partial orderings.

The behavior of a constraint node is as follows:

1. The node first executes its (optional) entry action. It then creates a table in which every event in the signature of a constraint associated with the node has a slot. (Each such event has a single slot in the table, even if it occurs in multiple constraints.) Each slot in the table contains three fields: the name of the event, the data associated with the event when it arrives, and a boolean variable that indicates whether the event arrived from the environment and did not cause a constraint violation. The data field of every slot is initially empty and the boolean variable in every slot is initially false.

2. The node sends a *prompt_req* event to the user interfaces (via the service monitor) – for every event e whose boolean variable is set to false in the table.

3. The node sends a *prompt_opt* event to the user interfaces (via the service monitor) – for every event e whose boolean variable is set to true in the table.

4. The node then waits for any event that is in the signature of any constraint associated with the node (i.e., has a slot in the table) or is the up-label of any ancestor node.

5. Upon arrival of any such event e, if e is the up-label of some ancestor node, control is transferred to that ancestor. Otherwise:

   (a) The boolean variable in the slot for e is set to true. The data associated with the event e is written in the table; if previous data are present, they are first erased.

   (b) The *enabled* constraints c are those that satisfy the following conditions:
      - The event e occurs in the signature of the constraint c.
      - All events in the signature of the constraint c have their boolean variables set to true in the table.

   (c) The enabled constraints c are evaluated in the specified priority order:
      - If the first/next constraint c in priority order is violated,
        - Its (optional) violation action and notify function are executed, and the returned *notify* event is sent to the user interfaces (via the service monitor).
        - The boolean variable in the slot for e is reset to false, and the data field is reinitialized to be empty.
        - The up-label function of constraint c is executed (if it is specified). If it returns the up-label of an ancestor node,
          - The up-label is generated and control is transferred to the ancestor node.
        - Else if the constraint has a specified child node,
          - Control is transferred to the specified node.
        - Else the constraint node goes back to waiting for events (step 2).
      - Else the next enabled constraint is evaluated. If none remain to be evaluated, the constraint node goes to step 5d.

   (d) If all enabled constraints were satisfied,
      - The (optional) satisfaction action and notify function of each satisfied constraint are executed, and the returned *notify* events are sent to the user interfaces (via the service monitor).
      - If all constraints associated with the node were enabled (and satisfied),
        - The (optional) exit action and notify function are executed and the returned *notify* event is sent to the user interfaces (via the service monitor).
        - The up-label function of the constraint node is executed (if it is specified). If it returns the up-label of an ancestor node,
          - The up-label is generated and control is transferred to the ancestor node.
        - Else if the constraint node has a specified child node,
          - Control is transferred to the specified node.
      - Else the constraint node goes back to waiting for events (step 2).

Table 5: The Execution of Constraint Nodes

6 An Example Execution of the Any-Time Teller

To illustrate the Sisl paradigm, we now describe an example execution of the Sisl Any-Time Teller depicted in Figure 4, using the web, automatic speech recognition, and telephone-based voice interfaces partially specified in Tables 2, 3 and 4.

Logging into the service

The service initially says "Welcome to the Any-Time Teller." The control point is at the root node, a constraint node. For the constraint to be satisfied, the login and the pin values must be identical (i.e. *login*==*pin*). The Sisl infrastructure automatically sends out a *prompt_req_login* and *prompt_req_pin* from the service logic to the user interfaces (via the service monitor). The user interfaces (via the prompt function) say "Please specify your login and personal identification number." For the web interface, text fields for the login and pin are automatically generated; for the speech-based interfaces, the grammars specified in the grammar function are automatically enabled.

In this scenario, suppose the user says "My login is Mary Smith and my pin is Mary Smith", and hence a *login* event with the value "Mary Smith" and a *pin* event with the value "Mary Smith" are sent to the service logic. Since the login and pin are identical, the constraint is satisfied. The Sisl infrastructure automatically sends out a *notify* event with the message "Hello Mary Smith. Welcome to the Sisl Any-Time Teller." The user interface says this message to the user.

Login successful

Control now proceeds to the choice node. The Sisl infrastructure automatically sends out

- *prompt_choice_startDeposit*,
- *prompt_choice_startWithdrawal*,
- *prompt_choice_startTransfer*, and
- *prompt_choice_userQuit*

events from the service logic to the user interfaces (via the service monitor), corresponding to the enabled choices. The user interfaces ask the user "What transaction would you like to do?" Figure 3 shows a screen snapshot of the web-based interface; the possible choices are automatically represented as radio buttons. For an automatic speech recognition interface, if the user says "I need help," the user interface says – via the help function of Table 2 – "You can make a withdrawal, deposit, transfer, or you can quit the service." Suppose the user now chooses to perform a transfer; the *startTransfer* event is sent to the service logic.

Transfer

Control now proceeds to the transfer constraint node. The Sisl infrastructure automatically sends out

- *prompt_req_src*,
- *prompt_req_tgt*, and
- *prompt_req_amt*

events from the service logic to the user interfaces (via the service monitor), together with a *up_MainMenu* event (since it is the up-label of an ancestor). Suppose the user responds with "I would like to transfer $1000 from my checking account," or equivalently "From checking, I'd like to transfer $1000." Either order of information is allowed; furthermore, this information is partial, since the target account is unspecified. The user interface sends a *src* and *amt* event, with the corresponding data, to the service monitor, which sends them one at a time to the service logic; suppose the *src* is sent first, followed by the *amt* event. The constraints `amt >=0`, `isValid(src)` and `amt <= Balance(src)` are automatically evaluated.

Suppose the checking account does not have a balance of at least $1000; hence, there is a constraint violation and the supplied amount information is erased since it was sent last. Note that constraints are evaluated as soon as possible: for example, the user is not required to specify a target account in order for the balance on the source account to be checked. The Sisl infrastructure then automatically sends out a *prompt_opt_src*, *prompt_req_tgt*, *prompt_req_amt*, and *up_MainMenu* event from the service logic to the user interfaces (via the service monitor), as well as a *notify* event with the message "Your checking account does not have sufficient funds to cover the amount of $1000. Please specify an amount and a target account." The user interface then notifies the user with this message and prompts the user for the information.

Suppose the user now says "Transfer $500 to savings." The *amt* and *tgt* events are sent to the service monitor, and passed to the service logic. The constraints are now all evaluated and satisfied, the service logic automatically sends a *notify* event to the user interfaces with the message "Your transfer of $500 from checking to savings was successful."

Control then goes back up to the choice node: the loop on the incoming arc to the choice node indicates that the corresponding subgraph is repeatedly executed until the condition on the arc becomes false. If the user wants to quit, the *userQuit* event is sent to the service logic, the hasQuit variable is set to true, and the loop is terminated.

Back up to the Main Menu

If the user would like to abort at any time during a withdrawal, deposit, or transfer transaction, he/she can say "I would like to go back up to the Main Menu," which results in an *up_MainMenu* event to be sent to the service logic. This causes control to return to the choice node, which has an associated upMainMenu label.

7 Other Sisl Features

We now describe two additional features of Sisl: a form of event priorities and lookahead, and a testing/debugging facility.

Event priorities and Lookahead As described in Section 3, the user interfaces collect a set of events from the user and dispatch them to the service monitor. The service monitor then selects and sends events from this set, one at a time, to the service logic. In particular, the service monitor will only send events that match the currently enabled set of events of the service logic. This behavior automatically supports a form of event priorities, since enabled events have higher priority than others. It also provides a form of lookahead, since additional information given by the user is stored until it becomes relevant. For example, consider a scenario of the Any-Time Teller in which the user has successfully logged in. The service logic's control point is at the choice node, and the enabled events of the service logic are *startDeposit*, *startWithdrawal*, *startTransfer*, and *userQuit*. Suppose the user now says "I would like to transfer $500 from checking," and hence the user interface passes the *startTransfer*, *src* and *amt* events to the service monitor. Since only the *startTransfer* event is enabled at this node, it automatically will be sent first by the service monitor. The service logic now proceeds to the transfer constraint node, and the *src*, *amt*, *tgt* events now become enabled. Since the user has already given the source and amount information, it will automatically be passed to the service logic, and the user will then be prompted for the remaining information: namely, the target account. In this fashion, Sisl services can automatically perform lookahead in the user input, without requiring any modifications to the service logic.

Specification-based testing Reactive constraint graphs in the Sisl language can be instrumented with a non-intrusive testing and debugging facility in the flavor of assert statements in traditional languages. In particular, service specifications can be expressed as safety properties, or conditions, on sequences of events in propositional linear-time temporal logic [Manna and Pnueli, 1992].

As a simple example, in the Any-Time Teller, the environment must send a *startTransfer* event (as represented by the event label on the choice node transition), before the *src*, *tgt*, and *amt* events can become enabled (as represented by the events in the transfer constraint node). In addition, the environment must first send *login* and *pin* events in any order to start up the service. This required behavior can be represented as a temporal logic specification; if this specified property is violated at any point during an execution of the service, the assertion fails and the Sisl toolset generates a special event. The service programmer/tester has the option to abort the application, ignore the failed assertion, or ask the system to report entire test traces.

8 Conclusions

Our implementation of the Any-Time Teller consists of approximately 500 lines of Sisl code. It currently has applet, HTML, automatic speech recognition, and VoiceXML [VoiceXML, 1999] interfaces, each about 300 lines, all sharing the same service logic.

We are currently using Sisl to develop an interactive service based on a system for visual exploration and analysis of data [Cox et al., 1999]. Our service supports multiple interfaces – including web, automatic speech recognition, and text-based interfaces, all with natural language understanding. All interfaces share the same service logic.

Moving further up in application size, Sisl is being used to prototype a new generation of call processing services for a Lucent Technologies switching product, as part of a collaboration between research and development. These services must be accessible via both graphical and speech-based natural language interfaces, and must support the inexperienced as well as expert user. As part of this collaboration, the research/development team has prototyped the requirements for some call processing features, and has integrated these prototyped services into the call processing platform of the actual switch. We have used web, touch-tone, and speech based interfaces to interact with these services.

We are also exploring the use of Sisl in the development of collaborative applications, in which users may interact with the system through a variety of devices including personal digital assistants, cellular telephones, and traditional desktop devices. For such applications, a richer class of information, including documents and and information visualization views, may be sent by the service logic to the user interfaces. We expect that this will lead to interesting extensions of Sisl, in which the transmitted information may be tailored for a variety of devices. For example, a spreadsheet or information visualization view may appear in a distilled form on a personal digital assistant, while a document may be automatically spoken on a cellular phone without a screen.

Detailed user studies and experience reports on these applications will appear in later publications.

Acknowledgments. Christopher Colby's and Radha Jagadeesan's work on Triveni is supported in part by a grant from National Science Foundation.

References

[Abella et al., 1996] 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.

[Atkins et al., 1997] Atkins, D. L., Ball, T., Benedikt, M., Cox, K., Ladd, D., Mataga, P., Puchol, C., Ramming, J., Rehor, K., and Tuckey, C. (1997). Integrated web and telephone service creation. *Bell Labs Technical Journal*, 2(1):19–35.

[Atkins et al., 1999] Atkins, D. L., Ball, T., Bruns, G., and Cox, K. (1999). Mawl: A domain-specific language for form-based services. *IEEE Transactions on Software Engineering*, 25(3):334–346.

[Brondsted et al., 1998] Brondsted, T., Bai, B., and Olsen, J. (1998). The Reward service creation environment, an overview. In *Proceedings of the International Conference on Spoken Language Processing*, volume 4, pages 1175–1178, Sydney, Australia. Australian Speech Science and Technology Association, Incorporated.

[Colby et al., 1998a] Colby, C., Jagadeesan, L. J., Jagadeesan, R., Läufer, K., and Puchol, C. (1998a). 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.

[Colby et al., 1998b] Colby, C., Jagadeesan, L. J., Jagadeesan, R., Läufer, K., and Puchol, C. (1998b). Objects and concurrency in Triveni: A telecommunication case study in Java. In *Proceedings of the 4th Conference on Object-Oriented Technologies and Systems (COOTS-98)*, pages 133–148, Santa Fe, New Mexico. USENIX Association.

[Cox et al., 1999] Cox, K., Hibino, S., Hong, L., Mockus, A., and Wills, G. (1999). Infostill: A task-oriented framework for analyzing data through information visualization. In *IEEE Symposium on Information Visualization Late-Breaking Results*, pages 19–22, San Francisco, CA. IEEE Computer Society.

[Francis et al., 1998] Francis, B., Fedorov, A., Harrison, R., Homer, A., Murphy, S., Smith, R., and Sussman, D. (1998). *Professional Active Server Pages 2.0.* Wrox Press Inc., Chicago, IL.

[Goddeau et al., 1996] 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.

[Hayes et al., 1985] Hayes, P. J., Szekely, P. A., and Lerner, R. A. (1985). Design alternatives for user interface management systems based on experience with COUSIN. In *Proceedings of Conference on Human Factors in Computing Systems*, Interface Tools and Structures, pages 169–175, San Francisco, CA. Association for Computing Machinery.

[Issar, 1997] 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.

[JSAPI, 1998] JSAPI (1998). Java Speech API programmer's guide. http://java.sun.com/products/java-media/speech/.

[Manna and Pnueli, 1992] Manna, Z. and Pnueli, A. (1992). *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York.

[McTear, 1998] McTear, M. (1998). Modelling spoken dialogues with state transition diagrams: experiences of the CSLU toolkit. In *Proceedings of the International Conference on Spoken Language Processing*, volume 4, pages 1223–1226, Sydney, Australia. Australian Speech Science and Technology Association, Incorporated.

[Myers and Rosson, 1992] Myers, B. and Rosson, M. (1992). Survey on user interface programming. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 195–202, Monterey, CA. Association for Computing Machinery.

[Pargellis et al., 1998] Pargellis, A., Zhou, Q., Saad, A., and Lee, C.-H. (1998). A language for creating speech applications. In *Proceedings of the International Conference on Spoken Language Processing*, volume 4, pages 1619–1622, Sydney, Australia. Australian Speech Science and Technology Association, Incorporated.

[Sutton et al., 1996] Sutton, S., Novick, D. G., Cole, R. A., Vermeulen, P., de Villiers, J., Schalkwyk, J., and Fanty, M. (1996). Building 10,000 spoken dialogue systems. In *Proceedings of the International Conference on Spoken Language Processing*, volume 2, pages 709–712, Philadelphia, PA. IEEE.

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

[Wang, 1998] 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.