# COMPONENT PROGRAMMING WITH OBJECT-ORIENTED SIGNALS

by

Sean McDirmid

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

December 2005

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

# SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Sean McDirmid

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: Wilson C. Hsieh

Matthew Flatt

Gary Lindstrom

Craig Chambers

Gail Murphy

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

# FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Sean McDirmid in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Wilson C. Hsieh
Chair: Supervisory Committee

Approved for the Major Department

Martin Berzins
Chair/Director

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

# ABSTRACT

Components that process state, rather than immutable data, are important because they make programs more usable with their ability to process continuously changing data. For example, a compiler built out of state-processing components can detect programming errors as soon as programmers write them. Unfortunately, state-processing components are difficult to assemble because existing languages depend on awkward event handling mechanisms to communicate changes in state. Because the encoding of event handling details is not very modular, glue code that assembles state-processing components is often excessively large.

This dissertation describes how state-processing components can be glued together with less code through time-varying values known as signals. Signals are state abstractions that hide event handling details from glue code by standardizing how state changes are communicated. We explore how signals can be used to assemble components in our language SuperGlue. Although signals have been explored in other languages, SuperGlue is unique in its use of simple wire-like connections to express component assemblies. To support components that process graph-like state, such as trees, SuperGlue has object-oriented abstractions that can be used to express an unbounded number of signal connections. With signals and objects, programmers can build complicated state-processing programs with reasonable amounts of glue code.

SuperGlue is a pragmatic language that supports the reuse of existing Java components. For example, SuperGlue code can reuse user-interface components in Java's Swing library. We have found that SuperGlue is beneficial in programs that involve significant amounts of state processing, such as many user-interface programs. For example, the SuperGlue implementation of an email client is about half the size of an equivalent Java implementation.

# CONTENTS

# PREFACE

Programming languages have advanced little in the last thirty years: the languages of today are only slightly better than the languages of yesterday. Although much of this has to do with the abilities and foresight of early computing pioneers, we are reaching a breaking point where we will require significantly better programming languages in the future. For this reason, in addition to exploring incremental improvements in existing languages, we must also explore new kinds of languages that radically change how programs are written. The research in this dissertation is my effort to do the latter.

I thank the many people on my committee, in the Flux Group, in Utah's PLT, and elsewhere who have helped me out through my years in graduate school. Specific thanks go to Patrick Tullman, Matthew Flatt, and Richard Cardone, who significantly inspired me with their arguments, ideas, and conversations in the early course of my research. I would especially like to thank my advisor, Wilson C. Hsieh, for his patience and focus, which made this dissertation possible.

# CHAPTER 1

# INTRODUCTION

Many kinds of components, such as user-interface widgets, continuously recompute their output data as their input data changes over time. State-processing components enhance program usability when compared to other kinds of components that must be re-executed manually whenever their input data changes. For example, compiler components can often only provide syntax and type errors for source code frozen at the time of a compiler execution. However, the compiler components of many development environments such as Eclipse [21] can recompute compiler errors for source code as it is being edited. In these compilers, programmers can be notified of compiler errors as soon as they type them. Because state-processing components automatically adapt to changing data inputs, they are valuable building blocks of programs that continuously aid users without requiring disruptive user attention.

The glue code that assembles state-processing components together is often difficult to write because existing programming languages lack good abstractions for communicating changes in state. Although many programming languages have good abstractions, such as fields, for storing and retrieving values in state, abstractions for communicating changes in the values of state often involve event-handling mechanisms. Writing glue code that deals with event handling involves managing how events are transmitted between components. This event-handling glue code is often difficult to write because it suffers from the following modularity problems:

– **Compatibility:** interfaces for handling events are often not standardized between components. As a result, glue code must often implement custom event handlers that adapt how events are handled between components. For example, in the Java implementation of an email client, the way an email folder transmits events about

new email messages must be adapted to how a user-interface table receives events about row insertions.

– **Redundancy:** dealing with event handling often involves encoding the same logic more than once. For example, in an email client, the logic of what email messages are being viewed must often be encoded twice: first to access the current email messages of a folder and second to detect when messages are added to or removed from the email folder.

– **Conditional use:** programs often refer to state in conditions that guard when and how components use state in other components. These conditions are often implemented as custom event handlers with a significant amount of redundant logic. For example, in an email client, a custom event handler must be implemented to express the condition of an email folder that is selected in a user-interface tree.

## 1.1 Example

We use the implementation of a user-interface program to demonstrate how state-processing components are difficult to glue together. A user-interface program in an object-oriented language is often organized according to a model-view controller [19] (MVC) architecture. Components in an MVC architecture process state by playing one of the following roles in a user-interface program:

– **models**, which encapsulate data stores such as databases or email mailboxes;

– **views**, which view state in models for the purpose of producing user output; and

– **controllers**, which view user input for the purpose of either updating models or modifying how user output is produced in views.

The MVC architecture promotes loose coupling between model, view, and controller components. Dependencies between MVC components can be configured externally in glue code, which deals with event handling to ensure that model, view, and controller components process each others' state correctly. For this reason, components in MVC

architectures are good examples of state-processing components whose glue code are difficult to write.

The specific user-interface program that we consider is an email client that glues together the following user interface and email components:

- A mailbox model component, which contains a hierarchy of email folders. Each folder contains a list of email messages.
- A tree-view component, which displays a hierarchy of nodes as output to users. This hierarchy of nodes is obtained from model components. Tree-view components also provide controller functionality: they support user input in the form of tree node selection.
- A table-view component, which displays a matrix of rows and columns as output to users.

Although the above list does not completely describe all components that are used in an email client, it is sufficient for our example. Our example focuses on how two requirements of the email client are implemented:

- A folder view, which is a tree-view component, should display as its nodes mailboxes and email folders. A list of mailboxes is viewed as the root nodes of the folder view. The children of a mailbox node are the top-level folders of the mailbox. The children of a folder node are the subfolders of the folder. The folder view of an email program is shown as the left tree widget in Figure 1.1.
- A message view, which is a table-view component, should display as its rows the email messages of an email folder that is selected in the folder view. If it is not the case that exactly one node is selected in the folder view, or if the selected node is not a folder, then the message view should be empty.[1] The message view of an email program is shown as the right table widget in Figure 1.1. Because the

[1] We use this behavior for example purposes only. Many email clients will merge the messages of multiple email folders that are selected in a folder view.

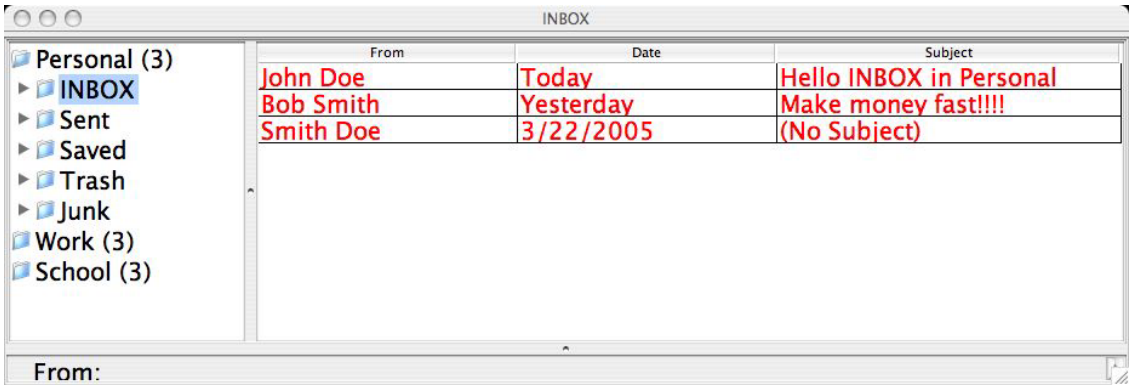


**Figure 1.1**. A screenshot of an email program; the folder view is the left tree widget in this screen; the message view is shown as the right table widget in this screen.

> `School` mailbox is selected in the folder view of Figure 1.1, the email messages of this folder are viewed in the message view.

In the rest of this section, we show how the glue code needed to implement these two requirements is complicated when Java Swing user-interface [40] and JavaMail [39] components are used. For simplicity, the following example uses APIs that are idealized versions of the APIs these libraries actually provide. If idealized APIs were not used, the examples shown here would be more complicated than is necessary to make our point.

Figure 1.2 and Figure 1.3 show Java glue code that implements the display behavior requirement of the folder view. The Java code in Figure 1.2 configures how current values of mailbox and email folder state are communicated to the folder view. The Java code in Figure 1.3 configures how changes in mailbox and email folder state are communicated to the folder view.

According to how the MVC architecture is implemented in Swing, tree views access current state by calling methods in `XXXModel` objects. In Figure 1.2, the model object of the folder view, `folderViewModel`, is an instance of the `AbstractTreeModel` class. The `getRoot()`, `getChild()`, and `getChildCount()` methods implemented inside this model object are called by the folder view when it views the current values of the state it is displaying. These methods are called when the folder view tree is repainted during program execution.

```
ArrayList<Mailbox> mailboxes = ...;
JTree folderView = new JTree();
AbstractTreeModel folderViewModel = new AbstractTreeModel() {
  Object getRoot() { return mailboxes; }
  Object getChildCount(Object node) {
    if (node instanceof List)
      return    ((List) node).size();
    if (node instanceof Mailbox)
      return ((Mailbox) node).getFolderCount();
    if (node instanceof Folder)
      return  ((Folder) node).getSubFolderCount();
  }
  Object getChild(Object node, int index) {
    if (node instanceof List)
      return    ((List) node).get(index);
    if (node instanceof Mailbox)
      return ((Mailbox) node).getFolder(index);
    if (node instanceof Folder)
      return  ((Folder) node).getSubFolder(index);
  }
};
folderView.setModel(folderViewModel);
```

**Figure 1.2**. Java glue code that implements and installs an email folder model object of a folder view component.

Swing and JavaMail both use the observer design pattern [18] to represent event handlers as objects that are managed flexibly in lists. According to how the MVC architecture is implemented in both Swing and JavaMail, observers that are used to view changes in state are objects of `XXXListener()` classes. Observer objects are installed and uninstalled by glue code with calls to `addXXXListener()` and `removeXXX-Listener()` methods on components that are being viewed. In Swing, components are notified of changes in state through calls to `notifyXXX()` methods on their model objects. The glue code in Figure 1.3 implements two observers: a folder observer, which views when folders are added to or removed from a mailbox or folder, and a tree expansion listener, which views when nodes are expanded and collapsed in a tree view component. In this code, the folder observer notifies the folder view of new child

```
FolderListener folderObserver =
  new FolderListener() {
  void folderAdded(Folder folder) {
    folderViewModel.notifyChildAdded
      (folder.getParent(), folder.getIndex());
  }
  void folderRemoved(Folder folder) {
    folderViewModel.notifyChildRemoved
      (folder.getParent(), folder.getIndex());
  }
};
folderView.addExpansionListener(
  new TreeExpansionListener() {
  void expanded(Object node) {
    if (node instanceof Folder)
      ((Folder) node).addFolderListener(folderObserver);
    /* Consider nodes of list and mailbox types. */
    else (node instanceof ...) ...;
  }
  void collapsed(Object node) {
    if (node instanceof Folder)
      ((Folder) node).removeFolderListener(folderObserver);
    else (node instanceof ...) ...;
  }
});
```

**Figure 1.3**. Java glue code that implements and installs the email folder observer objects of a folder view component.

nodes, and the tree expansion observer ensures that the folder observer is only installed on folders that are expanded nodes in the folder view.

The glue code in Figure 1.3 suffers from the following modularity problems:

– **Compatibility:** a custom folder observer must be implemented by the glue code in Figure 1.3 to translate email folder addition and removal events into tree view child addition and removal events. Swing, JavaMail, and many other popular Java libraries do not provide for any kind of event handling compatibility between components–they do not even provide reusable observer objects. Instead, event handling interfaces are very diverse and observer objects are always implemented in glue code.

– **Mode redundancy:** the glue code in Figure 1.3 encodes the same logic, which specifies what mailbox state is being displayed by the folder view, as the glue code in Figure 1.2: Figure 1.2 implements the logic for viewing current values, while Figure 1.3 implements the logic for viewing changes.

– **Transition redundancy:** the glue code in Figure 1.3 encodes twice the logic to deal with each of the following requirements:

  - When an email folder is added to or removed from a mailbox or folder, notify the folder view that a child node has been added or removed.
  - When a tree node that is a folder is expanded or collapsed, a folder observer is installed or uninstalled on this folder.

  Different interfaces are needed to accommodate both the beginning and end of a state configuration. For example, one set of interfaces is needed to express adding an element to a list, expanding a node, and installing an event handler on an event source, while another separate set of interfaces is needed to express removing an element from a list, collapsing a node, and uninstalling an event handler on an event source. Because these sets of interfaces are separated, glue code must encode the logic for each behavior twice.

– **Condition:** to minimize event-handling overhead, folder events are only handled on folders that are expanded in the folder view. As a result, the tree expansion listener in Figure 1.3 is implemented to install and uninstall the folder observer on folder nodes as they are expanded and collapsed.

The modularity problems demonstrated in Figure 1.3 are not uncommon in the glue code of an email client. Consider the Java code in Figure 1.4 and Figure 1.5, which implement the message view's model object and observer objects, respectively. As with the glue code in Figure 1.3, the glue code in Figure 1.4 and Figure 1.5 suffers from compatibility and redundancy modularity problems. These modularity problems are exacerbated by how the following two conditions are encoded: first, that only one node

```
JTable messageView = new JTable();
AbstractTableModel messageViewModel = new AbstractTableModel() {
  Folder active() {
    if (folderView.getSelectedCount() != 1) return null;
    if (!(folderView.getSelected(0) instanceof Folder))
      return null;
    return (Folder) folderView.getSelected(0);
  }
  int getRowCount() {
    if (active() != null)
      return active().getMessageCount();
    else return 0;
  }
  Object getDataAt(int row, int column) {
    Message msg = active().getMessage(row);
    ... // get column of msg as row.
  } ...
};
messageView.setModel(messageViewModel);
```

**Figure 1.4**. Java glue code that implements and installs the model object of a message view component.

of the folder view is selected, and second, that the only selected node is a folder. The glue code in Figure 1.4 can query the current value of these two conditions with only two **if** statements. However, as shown by the tree selection observer implementation in Figure 1.5, detecting changes in the values of these two conditions is very complicated. This object's implementation is complicated because it must detect and react to three boundary cases:

- When a node is added to or removed from the selection, and only one node that is a folder is selected, install the message observer on that selected node. This case actually combines two boundary cases, because there is only one minor syntactic difference between the implementations of these cases.

- When a node is added to the selection, the first selected node is a folder, and two nodes are selected, uninstall the message observer from the first selected node.

```
MessageCountListener messageObserver =
  new MessageCountListener() {
  void messageAdded(Message message) {
    messageViewModel.notifyRowInserted(message);
  }
  void messageRemoved(Message message) {
    messageViewModel.notifyRowDeleted(message);
  }
};
TreeSelectionListener selectionObserver =
  new TreeSelectionListener() {
  void selectionAdded  (Object node) {
    if (folderView.getSelected(0) instanceof Folder) {
      if (folderView.getSelectedCount() == 2) {
        ((Folder) folderView.getSelected(0)).
          removeMessageCountListener(messageObserver);
        messageViewModel.notifyRowsChanged();
      } else if (messageViewModel.active() != null) {
        messageViewModel.active().
          addMessageCountListener(messageObserver);
        messageViewModel.notifyRowsChanged();
      }
    }
  }
  void selectionRemoved(Object node) {
    if (folderView.getSelectedCount() == 0 &&
              node instanceof Folder) {
        ((Folder) node).
          removeMessageCountListener(messageObserver);
        messageViewModel.notifyRowsChanged();
    } else if (messageViewModel.active() != null) {
        messageViewModel.active().
          addMessageCountListener(messageObserver);
        messageViewModel.notifyRowsChanged();
      }
    }
  }
};
folderView.addSelectionListener(selectionObserver);
```

**Figure 1.5**. Java glue code that implements and installs the observer objects of a message view component.

– When a node is removed from the selection, the removed node is a folder, and no nodes are selected, uninstall the message observer from the removed node.

The implementations of these boundary cases exhibit a combination of the transition redundancy and condition modularity problems. As discussed in Section 1.2, this kind of problem is not dealt with very well using existing solutions.

## 1.2 Existing Solutions

Because modularity problems related to gluing state-processing components together are well known, various solutions have been designed to deal with these problems. Some of the modularity problems described in Section 1.1 can be dealt with through better component interfaces. Although the Swing and JavaMail libraries have professionally designed interfaces, perhaps not enough emphasis was placed on making these libraries easy to use. One potential area of improvement is in the use of shared interfaces to standardize how state is communicated between different kinds of components. Interface reuse is already encouraged in object-oriented languages. For example, Java's Collections Framework [37] provides standard interfaces for representing and manipulating collections. The `List` interface in the *`java.util`* package is a standard representation of a list, which can be used by different components to provide or receive lists. When one component provides a list through the `List` interface while another component receives a list through the `List` interface, these components are easier to glue together.

Unfortunately, Java's core libraries do not currently provide a `List` interface that standardizes how changes in list state are viewed. The `ObservableList` interface, which is declared in Figure 1.6, has been proposed [1] as an interface that can make user interfaces easier to write. The `ObservableList` interface supports the installation of list observers that can handle list element addition and removal events. Although custom list observers can be implemented in glue code to view changes in a list, the power of the `ObservableList` interface is only realized when different state-processing components use this interface. When components use observable lists, glue code does not need to implement list observers. Instead, glue code can exchange observable lists

```
interface ObservableList {
  int size();
  Object get(int index);
  void    addListObserver(ListObserver listener);
  void removeListObserver(ListObserver listener);
  interface ListObserver {
    void elementAdded  (Object element, int index);
    void elementRemoved(Object element, int index);
  }
}
```

**Figure 1.6**. The `ObservableList` Java interface, which describes a list with observable changes in membership.

between components, which lets components install their own list observers to view changes in these lists.

Figure 1.7 shows how the use of the `ObservableList` interface in Swing user-interface and JavaMail classes would eliminate more than two thirds of the code in Figure 1.2 and Figure 1.3. The model object of a user-interface tree component can implement the `getChildren()` method to returns node children as `ObservableList` objects. The `Folder` class can also contain a method `getSubFolders()` that returns the subfolders of a folder as an `ObservableList` object. With these enhancements, the glue code in Figure 1.7 only needs to call the `getSubFolders()` method on a folder object in the implementation of the folder-view model's `getChildren()` method. Using the generic `ObservableList` interface, all details of the necessary event handling can be dealt with through nonrepetitive component implementation code in the mailbox and user-interface tree components.

The standardized use of interfaces such as the `ObservableList` interface only sometimes solves the modularity problems described here. Standardized interfaces are only effective when data structures or behavior are generic enough to be used in multiple components. Also, standardized interfaces cannot by themselves solve the condition-modularity problem. For example in Figure 1.5, the implementation of the tree-selection observer cannot be replaced with the use of a standardized interface because tree node selection state is used conditionally. Standardized interfaces only reduce the amount

```
ObservableList mailboxes = ...;
JTree folderView = new JTree();
AbstractTreeModel folderViewModel = new AbstractTreeModel() {
  Object getRoot() { return mailboxes; }
  ObservableList  getChildren(Object node) {
    if (node instanceof ObservableList)
      return ((ObservableList) node);
    if (node instanceof Mailbox)
      return ((Mailbox) node).getFolders();
    if (node instanceof Folder)
      return ((Folder)  node).getSubFolders();
  }
};
folderView.setModel(folderViewModel);
```

**Figure 1.7**. Glue code rewritten from Figure 1.2 to use the `ObservableList` interface.

of code needed to view state between components, and are not effective in expressing conditions that refer to state.

Interfaces can also be designed to eliminate some forms of boundary redundancy in glue code. Rather than support begin and end-style events and operations with different interfaces, components can support events and operations with one interface that expresses the begin and end contexts as a parameter. For example, list `add()` and `remove()` methods can be replaced with a single `changed()` method that takes as an argument whether an element is added or removed as an argument. Although this technique can be effective, it suffers from three problems. First, programming to such an interface is unnatural for most programmers. For example, most programmers prefer having separate `add` and `remove` methods in an interface rather than a single `change` method with an extra boolean argument. Second, fewer methods with more arguments can make interfaces more difficult to use because programmers may have to understand arguments that handle cases not important to them. Third, this technique cannot eliminate kinds of boundary redundancies that do not completely share the same structure. For example, the code in Figure 1.5 that deals with tree selection and event handling would not benefit from this technique because handling selection and deselection events requires code with different structures.

Beyond better interface design, language design has also been used to address modularity problems in glue code. Many programming languages support callback procedures with a very verbose syntax. For example, expressing callbacks in Java is overly verbose because they are expressed as methods in classes. Code that glues together state-processing components often looks better in languages such as Smalltalk [19] or Python because they support callbacks with syntax that is less verbose. Although brief callback syntax can encourage reuse of small pieces of code through quickly defined high-order functions, it does not solve the modularity problems presented in this chapter.

Because the task of gluing together state-processing components often involves event handling, we could try enhancing programming languages with first-class event handling abstractions. Proposed event handling abstractions [31] provide a better syntax for expressing event handlers and firing events, but they do not address the modularity problems that we have discussed. Instead, to solve these problems, we must completely hide event handling details completely from glue code.

## 1.3 SuperGlue

This dissertation describes a methodology for building state-processing components that can be assembled together with less glue code. Components in our methodology are assembled together through *signals* [13], which represent state as time-varying values. Unlike conventional abstractions that represent state, such as fields, signals can be used directly in computations rather than their current values. A computation that refers to a signal is also a signal whose value changes automatically when the referred-to signal's value changes. For example, if `x` and `y` are signals, then `x + y` is a signal expression whose current value is always the sum of the current values for the `x` and `y` signals. Signals are abstractions that standardize how state is communicated between components. Although event handling can still be used to communicate changes in state through signals, the details of this event handling are no longer a concern of the glue code programmer.

We have designed and implemented a language known as SuperGlue where components are assembled together through signals. SuperGlue is based on a port and

connection paradigm. A component provides and receives data through ports that exist between the component's implementation and other components. Programmers then write glue code to connect the provided ports (*exports*) to the required ports (*imports*) of a program's components. The port-connection paradigm is often the basis for module systems [15, 27, 34] and architecture-description languages [2, 28] because connections allow for explicit and configurable component dependencies. For component assembly tasks, connections are often preferable to more powerful procedure abstractions, which obscure component relationships through recursion and indirection [24]. Because the emphasis of this dissertation is on glue code, signals in SuperGlue are manipulated through connections. SuperGlue's use of connections is in contrast to function applications, which are used to manipulate signals in functional-reactive programming languages [6, 13, 20].

Our design of SuperGlue overcomes the following two limitations of the basic connection-port paradigm:

- Connections are static relationships that cannot be changed during program execution.
- Expressing each connection individually does not scale, meaning a large or unbounded number of connections cannot be encoded effectively.

For many module systems and architecture-description languages, these two limitations are acceptable because connections in these languages are static and of a coarse grain. For example, connections in Jiazzi [27], which is a module system for Java, are static and manipulate packages that contain many classes. These two limitations must be addressed in SuperGlue because connections between state-processing components are often dynamic and of a fine granularity. For example, what is connected to a user-interface table's rows can change dynamically according to user selection, and user-interface trees contain a hierarchy of nodes that must be connected from another hierarchy of signals. To address these two limitations, connections in SuperGlue are augmented with two features:

– Connections in SuperGlue can be guarded by conditions whose truth values can change at run-time. Connections to the same port are prioritized in a *circuit* that switches how the port is connected at run-time based on connection conditions.

– Object-oriented abstractions are used to abstract over signal connections so connection graphs of large or unbounded sizes can be encoded in SuperGlue with nonrepetitive code.

These two features make SuperGlue code effective in gluing together state-processing components without the use of more powerful control flow and procedure constructs that obscure component relationships.

Circuits in SuperGlue are heavily related to signals: only signal expressions can guard a connection because dynamic switching depends on the encapsulated communication of changes in state. Circuits encode dynamic switching behavior in a static connection graph and enable the direct encoding of conditions that refer to state. For example, in the email client of Section 1.1, a circuit can change what email folder's messages are viewed in a message view table. Although circuits are not powerful enough to express arbitrary kinds of dynamic behavior, they can express the dynamic behavior that is commonly needed to assemble state-processing components together.

SuperGlue's object-oriented abstractions are used to construct a program's connection graph, which can be unbounded in size. Our approach uses extensible types in a way that is analogous to how virtual methods are dispatched in conventional object-oriented languages. An unbounded-sized connection graph is constructed through type-based pattern matching: new connections can be created relative to existing connections in the graph that match specified type patterns. Nodes in the connection graph are either objects or *inner objects*, which are the public members of objects that represent entities contained in these objects. For example, inner objects can represent the email folders of a mailbox and the nodes of a user-interface tree. An object can create an unbounded number of inner objects, and therefore inner objects can only be identified in connections by their types and by the types of the values that are connected to them. For example, a

connection can match each email folder that is connected to a tree node and then connect the folder's subfolders to the node's children.

Our object-oriented approach is based on how state-processing components are currently assembled in object-oriented languages such as Java, where run-time connection matching serves the same purpose as run-time type queries. Besides inner objects, SuperGlue supports class extension and interfaces, which are common abstractions in other object-oriented languages. These abstractions have been modified to work with connections rather than method calls. For example, class extension in SuperGlue is used to prioritize connections in circuits, which serves the same purpose as method overriding.

As a concrete example of how SuperGlue can reduce the amount of code needed to glue state-processing components together, consider the SuperGlue code in Figure 1.8. This code glues together the following state-processing components in an email client: a folder view (**`folderView`**), which is a user-interface tree component; an email mailbox (**`mailbox`**), which contains a hierarchy of email folders and messages; and a message view (**`messageView`**), which is a user-interface table component. The code in Figure 1.8 implements the following behavior: the messages of a folder are viewed as rows in a message view when only one folder is selected in a folder view. This code involves two conditions. The first line of Figure 1.8 checks that only one node is selected in the folder view. The second line of Figure 1.8 checks that the first and only node selected in the folder view is an email folder. This condition is expressed as a connection query, which checks if a value of a specified type is connected by a signal connection to the target value of the query. If both of these conditions are true, a connection on the third line of Figure 1.8 connects the `messages` signal of the email folder to the `rows` signal of the message view.

```
if (folderView.selected.size == 1 &&
    folder = <mailbox.Folder> folderView.selected.get(0))
  messageView.rows = folder.messages;
```

**Figure 1.8**. SuperGlue code that implements the following email client behavior: the messages of a folder are viewed as rows in a message view table when only one node is selected in a folder view tree, and this node is an email folder.

Because the SuperGlue code in Figure 1.8 is evaluated continuously, the two conditions in this code and the rows of the message view can change their values during program execution. The user could select more than one node in a folder view tree, which causes the first condition to become false, and then deselect nodes until only one node is selected, which causes the first condition to become true. The user can select a node in a folder view tree that is not an email folder, which causes the second condition to become false. When a new email message is added to the folder whose messages are connected to the message view table's rows, a new row is added to the message view table. All of this behavior occurs in SuperGlue because of three lines of code. By comparison, more than thirty lines of code are required in Figure 1.4 and Figure 1.5 to implement the same behavior in Java using Java's Swing and JavaMail libraries.

In addition to signals and object-oriented abstractions, SuperGlue has two pragmatic features. First, in addition to signals, SuperGlue supports imperative event and command streams, which can handle discrete states and operations, such as when a user-interface button is pushed or an email message is deleted. Streams in SuperGlue are designed to work with signals, e.g., signals can be used in conditions that guard stream connections and uses. Second, SuperGlue signals and streams can be used and implemented in Java code, which is useful for two reasons:

– SuperGlue lacks abstractions, such as recursive functions, for expressing algorithms that are often used in component implementations. Therefore, it is often easier to implement components in a full-featured language such as Java rather than SuperGlue.

– Existing Java class libraries can be wrapped by new SuperGlue class libraries. For example, we have implemented the GlueUI class library, which wraps classes in Java's Swing library.

The use of SuperGlue is beneficial in programs that involve significant amounts of continuous processing behavior. For example, we have found that the SuperGlue implementation of an email client contains about half as much code as the Java implementation of a similar client. The implementation of an email client benefits from SuperGlue

because it has navigation and viewing features that continuously process changing user input and email state. Using SuperGlue is not very beneficial for programming tasks that do not reuse state-processing components, such as in building user-interfaces that only produce output in response to explicit user commands. Instead, SuperGlue enables programmers to build highly usable programs out of state-processing components without having to deal with the complexity of event-handling.

## 1.4 Dissertation Overview

This dissertation describes SuperGlue and how it solves the glue code problems described in this chapter. The rest of this dissertation is organized as follows. Chapter 2 introduces SuperGlue with its signal, object-oriented, stream, and class implementation abstractions. Chapter 3 evaluates how SuperGlue improves on how state-processing components are glued together. This evaluation involves three case studies: the first study explores how a library is designed in SuperGlue, the second study compares the SuperGlue implementation of a email client with an implementation and Java, and the third study shows how SuperGlue can be used to build language-aware editors. Chapter 4 describes SuperGlue's semantics, syntax, and implementation. Chapter 5 discusses related work in the areas of functional-reactive, object-oriented, logic, constraint, and component programming languages. Chapter 6 summarizes our conclusions and describes future work.

# CHAPTER 2

# SUPERGLUE

The modularity problems that are described in Chapter 1 occur when glue code is exposed to event handling details that communicate state changes between components. When event handling is hidden from glue code with standardized interfaces, these modularity problems disappear and glue code is much easier to write. Unfortunately, standardized interfaces cannot hide event handling in many situations, such as when state is used in conditions.

This dissertation explores how language abstractions known as **signals** can be used as component connectors that hide event handling from glue code. Signals represent state directly as time-varying values. Our exploration involves designing a new language called SuperGlue with signals as its core abstraction. A SuperGlue program is assembled out of **objects**, which interact by viewing each others' state through signals. An object views state through its imported signals, and provides state for viewing through its exported signals. SuperGlue code defines program behavior by connecting the signals of the program's objects together.

SuperGlue code is organized into rule-like connections whose whose antecedents are conditions. Conditions can refer to signals and are recomputed when the values of these signals change. As a result, the use of state in conditions is directly expressible in SuperGlue code. Connections to the same signal are prioritized in a **circuit**, where only the highest priority connection whose conditions are currently true connects the signal. Because conditions can refer to signals, the connection chosen to connect a signal can change during program execution. This continuous switching behavior allows state to control program behavior without exposing glue code to event handling details.

SuperGlue has object-oriented abstractions to deal with state that is organized into graph-like structures of potentially unbounded sizes, such as lists and trees. Object-

oriented abstractions also enable the reuse of connection patterns in the same program and in multiple programs. Because circuit construction depends on being able to identify what signal is being connected, signals in SuperGlue can only be connected when their containing values can be statically identified. Object-oriented abstractions deal with this restriction by allowing glue code to abstract over signal-containing values by their types. As a result, these abstractions enable SuperGlue code to compactly express programs that can connect an unbounded number of signals. For example, object-oriented abstraction enables SuperGlue code to express the example in Section 1.1, where an email client's folder view tree has an unbounded number of mailbox and email folder nodes. SuperGlue's object-oriented abstractions are as follows:

- **Inner objects** have object-like declarations and are the public members of normal top-level objects. Inner objects are used to represent public entities that are created inside an object and can be unbounded in number.

- **Connection variables** abstract over signal connections according to the run-time types of inner objects and objects that are targeted by the connections. Connection variables allow multiple objects and inner objects to be connected based on their types, which is analogous to type-based method dispatch in object-oriented languages. Because inner objects are not created by the glue code that connects them, their ports can only be connected through connection variables.

- **Connection queries** can distinguish between inner objects based on what values are connected to them. The connection of other values to inner objects and the use of connection queries to query these values reifies existing connections to glue code. When inner objects, connection variables, and connection queries are used together, glue code can express new connections according to the structure of existing connections, which enables the construction of unbounded connection graphs.

- **Interfaces** are types that declare ports without implementing them. As in other object-oriented languages, when objects implement the same interfaces, they are

easier to glue together. Interfaces can also be used with connection variables to express mixin-like enhancements to existing objects and inner objects.

– **Extension** enables the reuse of object, inner object, and interface types. Connection overriding, which is analogous to method overriding, is supported through the use of type extension to prioritize signal connections in circuits. Extension in classes also enables the reuse of object implementations.

SuperGlue's abstractions are designed to help programmers deal with the complexity of program connection graphs in a way that is compatible with existing object-oriented programming styles and components. With SuperGlue's object-oriented abstractions, programmers can build explicit connection graphs of unbounded sizes in a way that is similar to how implicit connection graphs are constructed in existing object-oriented languages. For example, the use of email folders as user-interface tree nodes are often expressed in Java through model objects that dynamically associate tree node behavior with email folder values. In SuperGlue, this behavior can be directly expressed through email folder inner objects that are connected to tree node inner objects and are implemented with the same Java-based components.

Signals hide event-handling details from glue code by completely hiding control-flow details from glue code programmers. However, there are many situations where glue code programmers cannot avoid dealing with event-handling or control-flow details. For example, glue code might need to intercept an event where a user interface button is pushed, iterate over a list of email messages, or perform a command that deletes an email message. Because these events and commands involve discrete behavior, they cannot be effectively represented as signals. Instead, they are supported in SuperGlue through abstractions known as **streams**. Streams are designed to work effectively with signals so programmers do not need to sacrifice the benefit of using signals when they deal with discrete behavior.

Objects are instantiated from classes that are implemented in one of two ways. First, a class can be implemented by SuperGlue code that assembles other objects together. Class implementations in SuperGlue enable the reuse of glue code, so function-like

abstractions are not needed. Second, a class can be implemented in another programming language with a **driver** that maps between the language's abstractions and SuperGlue's abstractions. In our prototype of SuperGlue, classes can be implemented in Java according to a program architecture illustrated in Figure 2.1. In this illustration, two objects are instantiated from classes that are implemented in Java, and interact through circuits defined in SuperGlue code. Because of SuperGlue's object-oriented abstractions, SuperGlue classes can effectively represent many existing Java classes. For example, it is possible to use many widget classes in Java's Swing user-interface library as SuperGlue classes.

The rest of this chapter describes SuperGlue's design in detail. Section 2.1 describes SuperGlue's basic signal abstractions. Section 2.2 describes how SuperGlue signals are organized into objects with object-oriented abstractions. Section 2.3 describes how SuperGlue supports event and command stream abstractions. Section 2.4 describes how SuperGlue classes are implemented with SuperGlue and Java code. Section 2.5 discusses how SuperGlue's various language features are related.

## 2.1 Basic Signals

*Signals* [13] are abstractions that represent state as time-varying values. Conceptually, signals are evaluated continuously during a program's execution. In SuperGlue, continuous evaluation semantics are implemented with two mechanisms: a discrete mechanism that computes the signal's current value on request and a continuous mechanism that notifies viewers of changes in the signal's current value via event handling. Because these

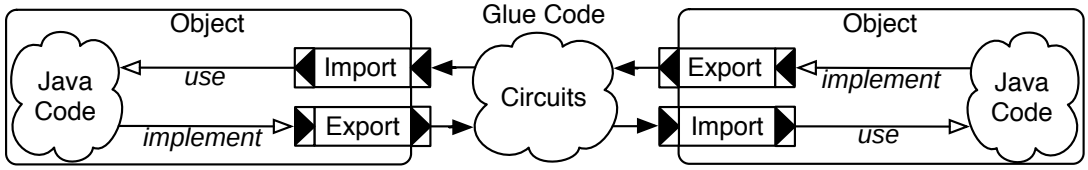


**Figure 2.1**. An illustration of a SuperGlue program's run-time architecture; "use" means the Java code is using imported signals through a special Java interface; "implement" means Java code is providing exported signals by implementing objects of a special Java interface.

two mechanisms are encapsulated inside signal abstractions, glue code programmers do not need to deal with control flow details such as event handling.

Objects, which are SuperGlue's units of assembly, interact by viewing each others' state through signals. Objects **export** signals to provide state that is viewed by other objects, and **import** signals to view state that is provided by other objects. An object is created from a class whose declaration specifies the object's imported and exported signals. Two example class declarations are shown in Figure 2.2. Class declarations in SuperGlue contain imported and exported signal declarations, where the type of a signal is expressed to the right of a colon. The `Thermometer` class declares an exported `temperature` signal that represents the temperature that thermometer objects measure. The `Label` class declares an imported `text` signal that represents the text that label objects display. The `Label` class also declares an imported `color` signal that represents the foreground color of a label object.

Two objects can interact in a program when an exported signal of one object is connected to an imported signal of the other object. SuperGlue's statement language, some of which is described in Figure 2.3, is designed to support the concise expression of signal connections between objects. SuperGlue code is contained inside class implementations, which are described in Section 2.4. For now, we assume that all SuperGlue code is expressed in one class with no imports or exports. Before objects are connected in SuperGlue, they must be declared with their instantiated classes. As an example of how objects are declared, consider the following object declaration:

```
let model = new Thermometer;

class Thermometer {
  export temperature : int;
}
class Label {
  import text  : String;
  import color : Color;
}
```

**Figure 2.2**. The declarations of the `Thermometer` and `Label` components.

```
statement : new | connection | if | block
new       : let object-id = new class-id;
connection: port-ref = port-ref;
port-ref  : object-id . port-id (args)?
if        : if ( condition ) statement |
            if ( condition ) statement else statement
condition : expression | condition && condition
expression: port-ref | object-id |
            constant | binary | ...
binary    : expression bin-op expression | ...
bin-op    : < | > | == | || | && | ...
block     : { statement* }
args      : ( expression (, expression)* )
```

**Figure 2.3**. The syntax of SuperGlue's statement language as described in this section; keywords and other terminals (except identifiers) are in bold; identifiers are in italics.

This code declares the `model` object as an instantiation of the `Thermometer` class.

Connections in SuperGlue are rules whose consequents are port connections and whose antecedents are conditions. Connection syntax in SuperGlue resembles assignments in a C-like language: the left-hand side of a connection is the port that is being connected, and the right-hand side of a connection is an expression that is connected to the signal. As an example of a connection, consider the following glue code:

```
let view = new Label;
view.text = "" + model.temperature;
```

This code connects the `temperature` signal exported from the **`model`** object to the `text` signal imported into the **`view`** object. Whenever the temperature measured by the **`model`** object changes, the text displayed in the **`view`** object is updated automatically to reflect this change. As in Java, SuperGlue supports the automatic conversion of integers to strings when combined with strings in plus expressions.

Conditions guard when connections are able to connect signals. When all the conditions of a connection evaluate to true, the connection is *active*, meaning that its source expression may be evaluated and used as the target signal's value. Condition syntax in

SuperGlue resembles C-like **if** statements. As an example of a condition, consider the following glue code:

```
if (model.temperature > 90)
  view.color = red;
```

This code connects the color red to the foreground color of the **view** object when the current temperature measured by the **model** object is greater than 90. When the current temperature is not greater than 90, the condition in this code prevents red from being the foreground color of the **view** object.

Although conditions and connections in SuperGlue resemble statements in an imperative language, they behave more like rules in a logic programming language. Conditions are evaluated continuously to determine if the connections they guard are active. In our example, the current temperature can dynamically go from being below 90 to being above 90, which causes the **view** object's foreground color to become red. In SuperGlue's implementation, this continuous evaluation is implemented through event handlers that activate the connection when the current temperature rises above 90.

### 2.1.1 Circuits

In SuperGlue, multiple connections can connect to the same signal. Together, these connections form a *circuit* that controls how a signal is connected during program execution. At any given time, any number of connections in a circuit can be active. If all connections in a circuit are inactive at the time that a signal is used, then a run-time error occurs. If exactly one connection in a circuit is active at some time, then the circuit's signal is connected according to that connection.

It is also possible that multiple connections in a circuit are active at the same time. Only one of these connections can connect the circuit's imported signal. To explicitly prioritize connections in a circuit, SuperGlue supports **else** clauses. Connections that are expressed in the body of an **else** clause are of a lower priority than connections expressed in the body of the corresponding **if** clause. As an example, the glue code in Figure 2.4 continuously connects a different color to the **label** object's foreground color depending on the current temperature. As the current temperature falls below 90,

```
let model = new Thermometer;
let view = new Label;
view.text = model.temperature;

if      (model.temperature > 90)
  view.color = red;
else if (model.temperature < 40)
  view.color = blue;
else
  view.color = black;
```

**Figure 2.4**. Glue code that glues together a **view** label object and **model** thermometer object.

the foreground color of the **view** label object changes from red to black. Likewise, as the current temperature falls below `40`, the foreground color of the **view** label object changes from black to blue.

The color connection code in Figure 2.4 forms a circuit that is illustrated in Figure 2.5. Conditions, which are triangles in the circuit, activate connection inputs (`in`) based on their test (`test`) inputs. The outputs of these conditions are connected to a switch with three prioritized input connections from highest priority (`hi`) to lowest priority (`lo`).

SuperGlue **else** clauses can only prioritize connections to the same signal. For example, two connections to different signals are not prioritized if they are separated by an **else** clause. This behavior differs from **if** and **else** clauses in imperative languages, where code in **else** clause is only executed if the condition of the corresponding **if** clause is false. Under certain circumstances in SuperGlue, a connection expressed in an **else** clause can connect a signal even if the conditions in the corresponding **if** clause are true.

Connections in a circuit do not always need to be prioritized with respect to each other. Prioritization might not be necessary because the conditions that guard different unprioritized connections are never true at the same time. Because there is no general way to determine if two connections can be active at the same time, run-time checks are used in SuperGlue to detect ambiguous connections. A run-time error that indicates an ambiguous connection occurs under the following conditions:

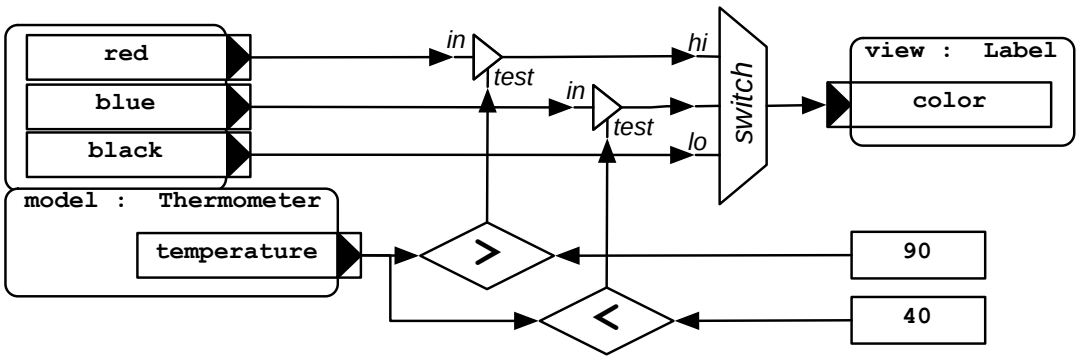


**Figure 2.5**. An illustration of a circuit that connects to the `color` signal of the **`view`** label object; rounded rectangles are objects; boxes that end with triangles are signals; inverter triangles are conditions; diamonds are tests; and squares are constants.

– A signal is currently being evaluated;

– Two active connections in the signal's circuit are not prioritized; and

– No other connection in the circuit is active with a higher priority.

The **`else`** clauses described in this section can only prioritize connections that are expressed in the same module. As described in Section 2.2.5, connections in different modules can be prioritized by the types of their containing objects. In this case, connections from separate modules can coexist in the same circuit without ambiguity through the use of natural object-oriented subtyping relationships.

## 2.2 Object-oriented Signals

With the basic port-connection paradigm, which is supported by the SuperGlue abstractions in Section 2.1, each connection in a program is encoded separately. This paradigm has two limitations:

– We cannot express programs that deal with stateful graph-like structures such as lists and trees. Graph-like structures are unbounded in their sizes and therefore cannot be manipulated with a fixed number of connections.

– Many connections conform to patterns that are repeated many times within the same program or across different programs. If each connection must be encoded separately, then these patterns cannot be modularized and connection code can become very repetitive.

These limitations effect the construction of real programs. Consider the email client used as an example in Section 1.1. This email client consists of a tree user-interface tree that displays a hierarchy of mailboxes and email folders. The limitations of the basic port-connection paradigm prevent this behavior from being expressed for two reasons:

– The layout of these mailboxes and email folders is unknown until and can change during program execution. As a result, the number of connections between email folders and user-interface tree nodes is not statically known.

– The kinds of connections needed between email folders and user-interface tree nodes conforms to a single pattern. For example, the subfolders of an email folder are connected to the children of a tree node when the email folder is connected to the tree node.

To address the two limitations of the port-connection paradigm, we enhance it with a mechanism to compute new parts of the connection graph according to the structure of existing parts. In SuperGlue, the types of the values in a connection graph are used to identify existing connections. SuperGlue supports such type-based pattern matching with object-oriented abstractions. For example, extension relationships are used to prioritize connections in circuits. The rest of this section describes SuperGlue's object-oriented abstractions.

### 2.2.1 Inner Objects

Before we can connect an unbounded number of connections, we must first be able to express connection graphs with an unbounded number of nodes. Such graphs can be expressed in SuperGlue with the *inner object* abstraction. Inner objects, which are members of objects, are similar to objects because they have ports and can undergo

extension. However, unlike top-level object, inner objects are not instantiated by glue code and therefore cannot be referred to by their individual identities. Instead, inner objects are created by objects to serve as constituent parts of the objects' interfaces. By creating inner objects, an object can have an unbounded number of ports, which simplifies the construction of connection graphs with an unbounded number of nodes. For example, a mailbox object can contain an unbounded number of email folder inner objects, and a user-interface tree can contain an unbounded number of tree node inner objects.

Inner object types, which are declared inside class declarations, specify the ports that are imported and exported by an inner object. The `TreeView` class, which is declared in Figure 2.6, contains the declaration of a `Node` inner object type that represents the nodes of user-interface trees. A node inner object imports three signals: the `text` signal that represents a node's display text, a `child_size` signal that represents the node's number of children, and a `child` signal that represents the node's indexed child nodes. An object of the `TreeView` class imports a `root` node signal and exports a `selected` node signal. The `child`, `root`, and `selected` signals are all of the `Node` inner object type, so the values that pass through these signals are either node inner objects or will be used to create node inner objects.

Inner objects are created in an object's implementation when the signals of an entity contained inside the object are accessed. For example, an object of the `TreeView` class in Figure 2.6 creates a node inner object the `child` signal of a node contained

```
class TreeView {
  inner Node {
    import text : String;
    import child_size : int;
    import child(index : int) : Node;
  }
  import root     : Node;
  export selected : Node;
}
```

**Figure 2.6**. The declarations of the `TreeView` class and the `Node` inner object type of the `TreeView` class.

in a user-interface tree is accessed. When an object creates an inner object, a value is connected to the inner object that distinguishes the inner object from other inner objects of the same type. The values that an object can connect to its inner objects can be expressed in the types of the object's imported signals. For example, in the `TreeView` class of Figure 2.6, the types of the imported `child` and `root` signals refer to the `Node` inner object type, so values obtained through these signals are connected to node inner objects. Consider the following SuperGlue code, which connects a mailbox object (whose class is declared in Figure 2.7) to the root node of a tree-view object:

```
let folderView = new TreeView;
let mailbox = new Mailbox;
folderView.root = mailbox.root_folder;
```

To display a tree of email folders, this code connects the root folder of a mailbox object to the root node of a folder-view object. The inner object types of the signals involved in this connection are incompatible: the root folder of a mailbox object is of the `Mailbox`'s `Folder` inner object type, while the root node of a folder view is of the `FolderView` class's `Node` inner object type. These two inner object types also do not export and import similar sets of signals. In a connection to an imported signals, inner object types do not have to match, and in fact should **never** match in reasonable programs; connecting an inner object to another inner object of the same class type is not meaningful because the former inner object becomes inaccessible. Instead, glue code must resolve the in-

```
class Mailbox {
  inner Message {...}
  inner Folder {
    export sub_folders_size : int;
    export sub_folder(i : int) : Folder;
  }
  export root_folder   : Folder;
}
```

**Figure 2.7**. The declarations of the `Mailbox` class and the `Message` and `Folder` inner object types of the `Mailbox` class.

compatibilities in these signal connections using the connection variable and connection query mechanisms that are described in Section 2.2.2 and Section 2.2.3.

### 2.2.2 Connection Variables

Glue code is responsible for connecting the imported signals of any inner object that is created by a program's objects. Glue code cannot connect the imported signals of a specific inner object because the identity of this inner object is encapsulated inside its creating object. Instead, glue code must connect the imported signals of multiple inner objects at once by using *connection variables*, which can abstract over both objects and inner objects by their types to connect their signals. Connection variables are declared in SuperGlue code according to the `connect-var` syntax in Figure 2.8. Connections expressed through a connection variable apply to all values of the connection variable's declared type. As an example, the following glue code connects the `children` imports of folder view nodes:

```
var (node : folderView.Node)
  node.child_size = 42;
```

This code declares the `node` connection variable that abstracts over node inner objects that are created by the folder-view object. Using the `node` connection variable as a target, the code connects `42` to every node inner object's imported `child_size` signal. As a result, all node inner objects are configured to have 42 children.

It is often useful to think of a connection variable as a universally quantified variable because the variable can be bound to any value of its specified type. Connection variables are only bound according to what connections are being queried, which is similar to how

```
statement  : new | connection | if | block | connect-var
connect-var: var ( var-id : var-type ) statement
var-type   : class-id | object-var . inner-id
object-var : var-id | object-id
port-ref   : object-var . port-id opt-args
```

**Figure 2.8**. The syntax of SuperGlue's statement language when considered with connection variables; this syntax builds on the syntax in Figure 2.3.

procedure arguments are only bound when their containing procedures are called. For example, the SuperGlue connection described in the previous paragraph provides the same functionality as the following Java code:

```
new TreeModel() {
  int getChildCount(Object node) {
    return 42;
  }
  ...
}
```

The `node` parameter of the `getChildCount()` method is only bound to a value when is method is called. This binding is similar to how a `node` connection variable is only bound when it is used to connect a `child_size` signal. Connection variables improve on method parameters in two ways. First, connection variables can be used to define the structure of an object or inner object without a verbose "shell" such as an inner class. Second, one connection variable can be used to connect multiple ports, while one method parameter can only be used in the definition of one method.

Because connection variables are bound only when port connections are resolved, they cannot be used to express iteration. Instead, iterator streams, which are described in Section 2.3.3, must be used to encode iteration in SuperGlue.

### 2.2.3 Connection Queries

To reify parts of the connection graph in SuperGlue code, values (constants, objects, and inner objects) can be connected to inner objects. Inner object connections are established through signal connections and are often indicated by the inner object type of an import. For example, when glue code connects the root folder signal of a mailbox to the root node signal of a folder view, an email folder inner object of the mailbox is connected to a tree node inner object of the folder view. The value that is connected to an inner object can then be queried in glue code, thus revealing existing signal connections. For example, SuperGlue code can query if a folder is connected to a tree node inner object, and act on this query to customize how the tree node's imports are connected.

Glue code can query what values are connected to inner objects with *connection queries*. A connection query is expressed as a condition in an **if** statement with the `query` syntax in Figure 2.9. A connection query tests if a target expression's evaluation is of a specified type or a value of this type is connected to the expression's evaluation. If a connection query succeeds, it is true and a value of the specified type is bound to the freshly declared variable. Because connection queries fail if values of their specified type are not connected to their target expressions, connection query variables are existentially quantified. The binding of a connection query variable is available to evaluate any conditions that follow the connection query, or any code that exists in the scope of the connection query's **if** clause. As an example, the following glue code queries if an email folder is connected to the tree node inner object that is bound to the `node` connection variable:

```
if (folder = <mailbox.Folder> node) ...
```

If some mailbox folder is connected to the node inner object that is bound to the `node` variable, the connection query succeeds, and this mailbox folder is bound to the `folder` variable. Otherwise, the `folder` variable is not bound and the connection that is guarded by this connection query is not true.

As mentioned in Section 2.2.1, signal connections that involve inner object types are always incompatible. These incompatibilities are resolved by using connection variables and connection queries together. Connection variables give glue code access to all inner objects of some type, while connection queries are used to customize connections to each of these inner object's ports based on what values are connected to the inner objects. As an example, the following glue code connects the `sub_folders_size` signals of

```
condition: expression | query | condition && condition
query    : var-id = < var-type > expression
```

**Figure 2.9**. The syntax of SuperGlue's statement language when considered with connection queries; this syntax builds on the syntax in Figure 2.8.

email folders to the imported `child_size` signals of the folder view object's node inner objects:

```
var (node : folderView.Node)
  if (folder = <mailbox.Folder> node)
    node.child_size = folder.sub_folders_size;
```

In our example, a folder inner object that is connected to a node inner object is used to provide the subfolders that are connected to the node inner object's children. This connection ensures that the number of subfolders in the folder inner object is used as the number of child nodes in the node inner object. Another connection connects every subfolder of a folder to a child node of the same index in the node that the folder is connected to:

```
var (node : folderView.Node, index : int)
  if (folder = <mailbox.Folder> node)
    node.child(index) = folder.sub_folder(index);
```

Because the children of a node inner object are also node inner objects, the folder view object connects folder inner objects to node inner objects whenever this connection is applied. As a result, this connection can be applied over and over again to create a hierarchy of signal connections. Although the entire connection graph that can be built through this connection is potentially unbounded in size,[1] termination occurs because connections are only computed for tree node inner objects that are currently displayed in the folder view.

Connection queries are similar in functionality and purpose to dynamic type checks and casts in Java. For example, in Figure 1.2 of Section 1.1, the following Java code is used to express how many children a tree node has:

```
Object getChildCount(Object node) {
    if (node instanceof Folder)
      return  ((Folder) node).getSubFolderCount();
    ...
}
```

[1] If every email folder contains ten email subfolders, then there are always tree nodes for the user to expand.

Tree nodes in Swing are generic objects in the basic model of a user-interface tree. Java code handles the case where an email folder is used as a tree node using a dynamic type check (**`instanceof`**) and a cast. This pattern is repeated in many existing Java components, especially those that conform to the MVC architectures. It is for this reason that we designed SuperGlue with inner objects, connection variables, and connection queries. For example, in SuperGlue, glue code can easily identify tree node inner objects that are connected from email folders.

The example described so far in this section expresses a list abstraction with two signals, e.g., the `child_size` and `child` signals are used to express a node inner object's list of child nodes. To make our example more realistic, we can use an interface to express a list abstraction as one signal. The `TreeView`, `Mailbox`, and `TableView` classes are declared in Figure 2.10 to refer to lists of node inner objects, email folders, and rows through signals of the `List` interface type. Interfaces are described in Section 2.2.4. For now, assume that the `List` interface declares a `size` signal to access the size of a list and a `get` signal to access the elements of a list.

The glue code in Figure 2.11 constructs the folder view and message view of an email client. Glue code for the folder view is similar to the glue code already presented in this section, except that a `List` interface reduces the number of connections to use a folder's subfolders as a node's children.

Because connection queries are conditions, they are evaluated continuously to determine if the connections they guard are active. When the expression tested by the connection query refers to a signal, the connection query's truth value and the value that it binds can change during program execution. As an example, consider how the last connection in Figure 2.11 connects the imported `rows` signal of a message view object, which is an object of the `TableView` class that is declared in Figure 2.10. The `TreeView` class in Figure 2.6 declares the exported `selected` signal, whose type is a list of node inner objects. The list of nodes that is selected in a tree view depends on the user, who can select and deselect nodes at any time during program execution. When a user selects only one node in the folder view tree, and this node is an email folder, the glue code in Figure 2.11 connects the email messages of this folder to the rows of the

```
class TreeView {
  inner object Node {
    import text : String;
    import children : List<Node>;
  }
  import root     : Node;
  export selected : List<Node>;
}
class Mailbox {
  inner object Message {...}
  inner object Folder {
    export messages    : List<Folder>;
    export sub_folders : List<Folder>;
  }
  export root_folder   : Folder;
}
class TableView {
  inner object Row    {...}
  import rows : List<Row>;
  ...
}
```

**Figure 2.10**. The declarations of the `TreeView`, `Mailbox`, and `TableView` class.

```
let messageView = new TableView;
let folderView = new TreeView;
let mailbox = new Mailbox;

folderView.root = mailbox.root_folder;

var (node : folderView.Node)
  if (folder = <mailbox.Folder> node)
    node.children = folder.sub_folders;

if (folderView.selected.size == 1 &&
    folder = <mailbox.Folder> folderView.selected.get(0))
  messageView.rows = folder.messages;
```

**Figure 2.11**. Code that connects the nodes of a folder view tree and the rows of a message view table.

message view table. As an email client executes, different nodes can be selected in the folder view tree, so different email messages can be displayed in the message view table.

SuperGlue's support for connection queries and other conditions that refer to signals solves many of the modularity problems discussed in Chapter 1. As a result, SuperGlue code can be significantly less complicated than Java code when gluing components together into an interactive program. For example, the last three lines of glue code in Figure 2.11 is equivalent in functionality to more than thirty lines of Java code in Figure 1.4 and Figure 1.5 of Section 1.1.

### 2.2.4 Interfaces

The next object-oriented abstraction that we discuss is the interface, which addresses the problem of reusing signal declarations in unrelated object implementations. For example, a list type consists of multiple signals, such as `size` and `get`, that are needed to represent a list. Redeclaring these signals every time that a list is used in a class definition is tedious, and different classes can possibly express lists with incompatible sets of signals. Also, it is often useful to abstract over objects and inner objects without referring to specific object types. For example, with a `Labeled` interface, glue code can express how a generic label signal is connected when an email folder is connected to any inner object in any object that imports the `Labeled` interface. Defining a label connection in this way is useful because there are many kinds of inner objects in different kinds of user-interface objects that support labeling.

SuperGlue interfaces are analogous to Java interfaces: they describe signals without describing the implementations of these signals. Interfaces do not specify if their signals are imported or exported, and the interface's signals can be used in either way in a class declaration. Interfaces in SuperGlue can be used in four ways: they can be extended by other interfaces (discussed in Section 2.2.5), they can be used in signal types, they can be imported into or exported by a class or inner object type, and they can be used as the declared types of connection or connection query variables.

Figure 2.12 shows the declarations of three interfaces: the `List` interface, which describes lists, and the `Labeled` interface, which describes objects that can have labels.

```
interface List<ENTRY> {
  port size : int;
  port get(index : int) : ENTRY;
  ...
}
interface Labeled {
  port text : String;
  ...
}
```

**Figure 2.12**. The declarations of the `List` and `Labeled` interfaces.

Signals in an interface are declared with the **`port`** keyword, and whether the signal is imported or exported is not specified. The `List` interface also has a type variable that describes the list's entries. Type variables in SuperGlue are similar to type variables in Java, which are erased before program execution.

If an interface is used as the type of an imported signal, then the interface's signals are imported in that class. For example, signals of the `List` interface are imported under the `children` signal in the following declaration of the `TreeView` class:

```
class TreeView {
  inner object Node {
    import children : List<Node>
    ...
  }
  ...
}
```

If an interface is used as the type of an exported signal, then the interface's signals are exported in that class. For example, signals of the `List` interface are exported under the `sub_folders` signal in the following declaration of the `Mailbox` class:

```
class Mailbox {
  inner object Folder {
    export sub_folders : List<Folder>;
    ...
  }
  ...
}
```

When two signals of the same interface are connected together, the signals of this interface are connected automatically between these two signals. As an example, consider the following SuperGlue code:

```
var (node : folderView.Node)
  if (folder = <mailbox.Folder> node)
    node.children = folder.sub_folders;
```

The above connection from a `sub_folders` signal to a `children` signal automatically implies the following connections between signals in the list interface:

```
node.children.get = folder.sub_folders.get;
node.children.size = folder.sub_folders.size;
```

In this way, interfaces not only enable the reuse of signal declarations between different classes, they also make objects easier to glue together. Interfaces are especially important when declaring inner object types. Inner object types cannot be shared between unrelated classes because their implementations are encapsulated in their containing classes. If two inner object types in different classes share a similar set of signals, the use of interfaces is the only way to express this similarity in SuperGlue

Interfaces can be used as connection variable types to abstract over objects and inner objects without referring to specific objects or classes. This usage enables glue code to express what happens when an object is connected to a general kind of inner object. For example, the following glue code specifies how an email folder should be labeled:

```
var (labeled : Labeled)
  if (folder = <mailbox.Folder> labeled)
    labeled.text = folder.name +
      " (" + folder.messages.size + ")";
```

This glue code declares a connection variable `labeled` of the imported `Labeled` interface type, meaning it abstracts over objects or inner objects that import the `Labeled` interface. The `labeled` connection variable is used as the target of a connection query. If an email folder is connected to an inner object that imports the `Labeled` interface, the name of the folder its the number of messages is connected to this object's `text`

signal. Since tree nodes have labels, they can also implement the `Labeled` interface to take advantage of connections to signals in objects of the `Labeled` interface:

```
class TreeView {
  inner object Node imports Labeled { ... }
... }
```

As a result, whenever an email folder is connected to a tree node, the text label of the tree node displays the folder's name and its number of messages.

### 2.2.5 Extension

Class extension in object-oriented languages allows a class to simultaneously inherit the type and implementation of another class. To support type reuse, implementation reuse, and the wrapping of classes written in conventional object-oriented languages, SuperGlue classes, inner object types, and interfaces also support extension. The semantics of SuperGlue class and inner object extension is analogous to Java class extension: classes inherit the imports, exports, inner objects, and implementations of extended classes. Likewise, the semantics of SuperGlue interface extension is analogous to Java interface extension. Extension enables polymorphism in SuperGlue, where the values of a type can be used in connections as values of the type's extended types. Additionally, connection variables declared with a type abstract over all values of that type, which includes values of extending types.

As an example of how class and inner object extension is used, consider the user interface classes declared in Figure 2.13. The `Widget` class is extended by all user interface classes. The `Widget` class declares signals, such as `tooltip`, that all widgets export. The `Container` class is extended by all widgets that contain other widgets. All containers inherit an `Entry` inner object type that represents container elements. Values connected to a container entry must be widgets, which is expressed as a constraint on the `Entry` inner object type with the **`isa`** keyword. Both the `Widget` and `Container` classes are abstract, which means they are only meant to be extended by other classes and cannot be used to create objects. The `SplitPane` class declared in Figure 2.13 is a concrete container that separates two widgets (`left` and `right`) with a movable

```
abstract class Widget {
  export tooltip : String;
}
abstract class Container extends Widget {
  inner object Entry isa Widget {
    import scrollable : boolean;
  }
}
class SplitPane extends Container {
  import left  : Entry;
  import right : Entry;
}
class  TreeView extends Widget {...}
class TableView extends Widget {...}
```

**Figure 2.13**. The declarations of the `Widget`, `Container`, `SplitPane`, `TreeView`, and `TableView` classes.

divider. The `TreeView` and `TableView` classes, originally declared in Figure 2.6 and Figure 2.10, are redeclared to be widgets.

With extension, objects can be used polymorphically. As an example consider the following code:

```
let folderView = new TreeView;
let messageView = new TableView;
let pane = new SplitPane;
pane.left  =  folderView;
pane.right = messageView;
```

The expression "`[folderView, messageView]`" is a list whose entries are the **`folderView`** and **`messageView`** objects. Because the `TreeView` and `TableView` classes both extend the `Widget` class, the **`folderView`** and **`messageView`** objects are both widgets that can be connected to the `left` and `right` imported signals of a split pane container.

Class, inner object, and interface extension are involved in the prioritization of connections in circuits. As described in Section 2.1.1, connection priorities act to resolve ambiguities when more than one connection is active in a circuit at the same time. The prioritization described in Section 2.1.1 is locally specified in glue code with **`else`**

clauses. However, a form of global prioritization is also needed to deal with connections that are not written together and do not exist in the same source code file. In SuperGlue, this global prioritization is based on class, inner object, and interface types. Extension relationships are used to determine how specific the target type of a connect is to the run-time type of a value whose connection is being queried. Connections with types that are more specific than other connections have a higher priority than these connections.

Rules for prioritizing connections based on types are presented in Section 4.2. Informally, connections to the ports of specific objects have a higher priority than connections that target connection variables. Otherwise, extension and implements relationships are used to prioritize different connections expressed through connection variables.

As an example, consider the following glue code:

```
var (table : TableView)
  table.rows = [];

if (folderView.selected.size == 1 &&
    folder = <mailbox.Folder> folderView.selected.get(0))
  messageView.rows = folder.messages;
```

The target of the first connection is the `table` variable with the `TableView` type while the target of the second connection is the **`messageView`** instance. Because objects are more specific than variables with class types, the second connection has a higher priority than the first connection. As a result, the message view table is connected to an empty list (`[]`) only when the first entry of the folder view tree's selected list does not exist or is not a folder. In other words, the first connection expresses "default" behavior that applies to all table view objects, while the second connection "overrides" the first connection for a specific table view object in a specific situation.

Because connections can be prioritized by extension relationships, object and inner object behavior can be overridden in an object-oriented way. As an example, consider the following SuperGlue code:

```
class Bird {
  import flies : boolean;
}
class Penguin extends Bird {}
```

```
var (bird    : Bird   ) bird   .flies = true;
var (penguin : Penguin) penguin.flies = false;
```

This code expresses that in general birds can fly but penguins, which are also birds, cannot. The first connection in this code applies to all birds and is therefore of a lower priority than the second connection, which only applies to penguins.

An additional criterion prioritizes connections based on how connection variables are used to bind other variables in connection queries. This criterion deals with cases where values of compatible types are connected to inner objects. As an example, consider code that defines two connections that connect `scrollable` signals imported into container entries:

```
var (entry : Container.Entry) {
  if (widget = <Widget>   entry) entry.scrollable = false;
  if (tree   = <TreeView> entry) entry.scrollable = true;
}
```

The first connection disables scrolling for container entries that widgets are connected to. The second connection enables scrolling for container entries that tree views are connected to. Because the target of both connections is an `entry` connection variable, the `Widget` and `TreeView` class types are compared when prioritizing the connections. Because the `TreeView` class extends the `Widget` class, the second connection in this code has a higher priority than the first.

The type-based prioritization of connections described here is analogous to virtual method dispatch in conventional object-oriented languages. Overriding in SuperGlue involves specifying a new connection with a target whose type is more specific than existing connections to the same signal. Finally, the ability in SuperGlue to prioritize connections based on the types of values that are connected to inner objects is analogous to multiple dispatch in languages that support multimethods [4].

Because arbitrary conditions can be used to guard connections in SuperGlue, it is difficult and often impossible to determine statically if a signal can be unconnected or connected ambiguously when the signal can be used. Detecting ambiguity is also an issue in other object-oriented languages that support predicate dispatch, which can suffer from

*method-not-understood* or *message-ambiguous* errors [14, 29]. Although conservative algorithms exist for detecting these errors statically, they often depend on global analyses or limit the power of expressions that guard connections. For these reasons, dynamic type checking is used in SuperGlue to detect type ambiguity.

## 2.3 Streams

Signals are abstractions that hide program control flow details from glue code programmers. Although the use of signals avoid many kinds of modularity problems in glue code because they hide control flow, many kinds of program behavior that should be expressed in glue code deal with control flow. Consider the following email client behavior as an example of how signals are limited: pushing the delete button should cause all email messages selected in the message view table to be deleted. This behavior involves three kinds of abstractions that cannot be represented by a signal. First, it involves an **event** abstraction that indicates a time instant when a button is pushed. Second, it involves a **command** abstraction that deletes an email message at a specified time instant. Third, it involves an **iterator** abstraction that iterates over multiple email messages that are selected in the message view table. Although this message deletion behavior could be expressed in Java code, doing so would be very inconvenient because the code required to switch between SuperGlue and Java would dominate the complexity of this task. Because this behavior involves assembling components together, it must be possible to implement it in an elegant way with SuperGlue code.

To manipulate control flow details, SuperGlue supports *streams*, which abstract over events and commands. Events occur while commands are performed at discrete times in a program's execution. SuperGlue streams can be used with signals without sacrificing the programming benefits of using signals. Streams come in two basic flavors: *event streams* that are used to transmit control and data from one caller to multiple callees and *command streams* that are used to transmit data and control from multiple callers to one callee. Event and command streams are described in Section 2.3.1. Event and command streams can be used to create *closures*, which can be used to freeze (capture) variable bindings, create objects, and sequence command execution. Closures are described in

Section 2.3.2. Event and command streams are also used in *iterator streams*, which abstract over sets of data. Iterator streams are described in Section 2.3.3. Finally, Section 2.3.4 describes how streams are used with signals in SuperGlue code.

### 2.3.1 Events and Commands

Event and command streams enable the direct manipulation of control flow events in SuperGlue code. When compared to a conventional programming language, command streams are analogous to procedures with multiple possible callers and exactly one callee, while event streams are analogous to callbacks with exactly one caller and multiple possible callees. Although event and command streams involve similar mechanics, they are used in different ways: command streams are used to convey that something should be done, while event streams convey that something has occurred. For example, command streams can be used to discretely update program state, while event streams can be used to express the occurrence of an error in a component.

Like signals, streams are ports in objects and inner objects. The syntax of a stream declaration is similar to the syntax of a signal declaration: a stream declaration is imported or exported, can have arguments, and can have a type. Unlike signals, the type of a stream can be **`void`**, meaning the event or command stream does not return data when used. The `Button` and `Mailbox` class declarations in Figure 2.14 are examples of how streams are declared. The `Button` class is a user interface control that declares an exported `pushed` event stream. When a user pushes a button, an event is transmitted through the button's `pushed` event stream. The `Message` inner type of the `Mailbox` class describes an email message that exports a `delete` command stream. To delete an email message from its mailbox, a command can be transmitted through the email message's `delete` command stream.

The syntax for using streams in glue code is shown in Figure 2.15. Event streams are accessed in **`on`** statements, while command streams are accessed in **`do`** statements. Both **`on`** and **`do`** statements can be guarded by conditions expressed in **`if`** statements. Because they can be guarded by conditions, **`do`** and **`on`** statements together often resemble Event-Condition-Action (ECA) rules. As an example of how **`do`** and **`on`** statements are used,

```
class Button extends Widget {
  export event pushed; ...
}
class Mailbox {
  class Message {
    export command delete : void; ...
  } ...
}
```

**Figure 2.14**. Declarations of the `Button` and `Mailbox` classes.

```
statement: new | connection | if | block | connect-var |
           on | do
on       : on ( port-ref ) statement |
           on ( var-id = port-ref ) statement
do       : do   port-ref ; |
           do ( port-ref ) statement |
           do ( var-id = port-ref ) statement
```

**Figure 2.15**. The informal syntax of the SuperGlue statement language when considered with streams; this syntax builds on the syntax in Figure 2.9.

consider the following SuperGlue code that deletes an email message when a delete button is pushed:

```
let delete_button = new Button;
/* message is a variable bound to an email message */
on (delete_button.pushed)
  do message.delete;
```

The **on** statement in the above glue code receives an `pushed` event from the delete button (**delete_button**) when it is pushed. When the delete button is pushed, the **do** statement in the above glue code sends a `delete` command to the email message that is bound to the `message` variable.

Imported streams are connected in circuits in the same way that imported signals are: stream connection can be guarded by conditions and can be prioritized using **else** statements or type specificity. Only an exported stream of the same kind can be directly connected to an imported stream. To connect a command stream to an event stream or

vice versa, a special bridge object must be interposed on the connection. For example, the `Function` class in SuperGlue's core library can be used to create function objects that fire events when commands are performed.

Both event and command streams return values when their types are not void. These return values are bound to a freshly declared variable. For example, the following code gets the next random number from a random number factory:

```
let random = new Random;
do (rnd = random.next)
  do sys.println("Random number is " + rnd);
```

This code also demonstrates how a **do** statement can specify statements that are executed after the **do** statement's own execution. While **on** statements will always have statements that are evaluated when an event occurs, these successive statements are optional for **do** statements.

### 2.3.2 Closures

The statement bodies of all **on** and **do** statements are closures with two important properties. First, a closure inherits an evaluation context that holds variable bindings that are in effect when the closure is created. A closure uses the variables bindings of this evaluation context, which foregoes the referential transparency behavior of nonclosure SuperGlue code. Note that the conditions that guard a closure's creation could become false after the closure's creation. Second, the creation of a closure is an imperative operation that can lead to the creation of new objects. Connections specified inside a closure's definition can then connect the imports of these new objects.

As an example of how closures work, consider the following code that creates a label when a button is pushed:

```
let button = new Button;
let sys = new System;
let time = sys.currentTime;
on (my_button.pushed) {
  let label = new Label;
  label.text = "Time is " + time;
}
```

In this code, the variable `time` is declared and bound to the value of a time signal (**sys**.time) using the *let* syntax. When the button created by this code is pushed, a closure is created with a label whose text displays the time that the button was pushed. For example, if the button is pushed at `4:32`, the string displayed by the label is `"Time is 4:32"`. If the button in this code is pushed multiple times, multiple labels are created that each display the time of a different button push. As a contrasting example, consider this code with the **let** statement moved to inside the **on** statement:

```
let button = new Button;
let sys = new System;
on (my.button.pushed) {
  let time = sys.currentTime;
  let label = new Label;
  label.text = "Time is " + time;
}
```

In this code, the labels that are created when the button is pushed always display the current time because the **let** statement undergoes continuous evaluation with respect to the context in which the closure's connections are considered. Only variables that are frozen by a closure do not undergo continuous evaluation inside the closure.

The body of a **do** statement is a closure that is created after the specified command finishes executing, which enables the sequencing of commands and provides access to the command's return value (if any). A **do** statement executes if its conditions are true when its enclosing closure is created. For top-level **do** statements their enclosing closure is the containing object.

Streams cannot be accessed in a context where a connection variable is defined because a connection variable cannot be bound by a stream access. Only objects that are created within a closure can be connected inside the closure. This means that any connection variables that are defined inside the closure apply only to objects created by the closure or other closures that contained by the closure.

When a closure is created, the closure creates and initializes any objects that are declared inside the closure, executes the **do** statements that it contains, and enables **on** statements so that they can receive events. The order that objects are initialized and **do**

statements are performed depends on the lexical order of these operations in the closure's definition.

### 2.3.3 Iterators

Iterator streams are time-varying concrete sets of values. Unlike normal signals, iterator streams cannot be used to create connections; instead, they can only be used in stream accesses. The syntax for using an iterator stream is shown in Figure 2.16. An iterator stream is accessed with the **for** statement, which causes each of its elements to be bound to a new variable. This variable can be used in the expression of another stream access, which is performed (command stream) or enabled (event stream) for each element of the iterator stream

When an iterator stream is used in a command stream access, a separate closure is created for each of the iterator stream's elements. As an example, the following glue code deletes all messages selected in a message view object when a delete button is pressed:

```
on (delete_button.pushed)
  for (row = messageView.selected.all)
    if (messages = <mailbox.Message> row)
      do message.delete;
```

The exported `selected` signal, which is declared in the `TableView` class of Figure 2.10, represents a list of rows that are current selected in a table view. The `List` interface, which is declared in Figure 2.17, declares an `all` iterator stream that represents every value currently in the list. To iterate over all rows that are currently selected in a message view table, the `all` iterator stream is referenced in a **for** statement, which binds each row selected in the message view table to the `row` variable in separate closures. In each closure, this code then deletes a message that is connected to the row.

```
statement: new | connection | if | block | connect-var |
           on | do | for
for: for ( var-id = port-ref ) statement
```

**Figure 2.16**. The syntax of SuperGlue's with iterator streams; this syntax builds on the syntax in Figure 2.15.

```
interface List<ENTRY> {
  port size : int;
  port get(index : int) : ENTRY;
  port next(entry : ENTRY) : ENTRY;

  port iterator all : ENTRY;
  ...
}
```

**Figure 2.17**. The signature of the `List` interface that demonstrates iterator streams; this is an expanded version of the `List` interface declared in Figure 2.12.

When an iterator stream is used in an event stream access, the event stream access receives events for every element of the iterator stream. When an element is added to an iterator stream, the variable of the **`for`** statement is bound to the added element and the targeted event stream can receive events for this binding. When an element is removed from an iterator stream, the variable of the **`for`** statement is bound to the removed element and the targeted event stream can no longer receive events for this binding. As an example of how iterator streams are used in event stream accesses, consider the following SuperGlue code:

```
for (element = list.all)
  if (button = <Button> element)
    on (button.pushed)
      do sys.println(button + " pushed");
```

In the above code, when a button exists in a **`list`** object, pushing that button will cause a message to be printed. The message will not be printed if one of the following is true: the button has been removed from the list, or the button has not yet been added to the list.

When the keywords **`begin`** and **`end`** are used under the scopes of **`for`** statements, glue code can directly intercept activation events and deactivation events. As an example, the following code prints text to the console when a message is selected or deselected:

```
for (row = messageView.selected.all)
  if (messages = <mailbox.Message> row) {
    on (begin) do sys.println("  selected: " + message);
    on (end  ) do sys.println("deselected: " + message);
  }
```

As described in Section 2.3.4, **begin** and **end on** statements are also used to extract events from signals.

Iterator streams are similar to types only in that they both represent sets of values. A type represents an immutable set of values that cannot be iterated over, while an iterator stream represents a mutable set of values that can be iterated over. Because of these differences, connection variables, which access types, and iterator stream accesses are used for completely different purposes. Connection variables are used to connect ports for all values of some type, while iterator stream accesses are used to iterate over all values in a set.

Our current design of SuperGlue lacks the aggregate operators that would allow iterator streams to be accessed as signals rather than as lower-level command and event streams. In other words, **for** statements cannot guard connections. In the future, we plan to explore how **for** statements can guard connections, which would allow for the filtering and redirection of iterator streams. This future work is described in Section 6.2.

### 2.3.4 Integration with Signals

Using streams to glue components together does not significantly improve on how similar glue code operations are expressed in other existing languages. If only streams are used to glue together components, there is no significant advantage to using SuperGlue. The benefit of using streams comes from their integration with signals: **on**, **do**, and **for** statements can be guarded by conditions that refer to signals, signals can be connected in closures, and there is no need to switch languages when dealing with streams instead of signals. Streams also form the lower-level parts of a class: a signal is defined in terms of a command stream and an event stream. The command stream is used to get the signal's current value, while the event stream is used to detect changes in the signal's value.

A signal's command stream can be accessed through a **let** statement, where the resulting variable binding is immediately frozen in a closure. A signal's event streams are accessed in SuperGlue code using the closure creation and destruction semantics that are used with iterator streams. A condition can form a closure that is created when the condition becomes true, and is destroyed when the condition becomes false. **on**

```
/* list is bound to some list */
/* comparator is bound to some comparator */

for (entry = list.all)
  if (entry.index < list.size - 1) {
    let next = list.get(entry.index + 1);
    if (comparator.compare(entry, next) > 0) {
      on (begin) do unsorted_count.increment;
      on (end)   do unsorted_count.decrement;
    }
  }
}
if (unsorted_count.result == 0) ... /* list is sorted. */
```

**Figure 2.18**. SuperGlue code that detects if a list is sorted.

(**begin**) and **on** (**end**) clauses can then be used to detect when this closure is created or destroyed. As an example, consider the following code:

```
if (comparator.compare(entry0, entry1) > 0) {
  on (begin) do unsorted_count.increment;
  on (end)   do unsorted_count.decrement;
}
```

This code uses a comparator to compare two values bound to the `entry0` and `entry1` variables. When they become unsorted, a **begin** event is transmitted and the **unsorted_count** is incremented by one. When they become sorted, an **end** event is transmitted and the **unsorted_count** is decremented by one.

As an example of how signals and streams are used together, consider the SuperGlue code in Figure 2.18, which detects if a list bound to the `list` variable is sorted. This code uses the `all` iterator stream declared in the `List` interface (Figure 2.17) to detect when an entry is added to or removed from the list. The `compare` signal of the `comparator` variable is then tested in a condition, which checks if the entry is sorted with respect to its successor. **on** (**begin**) and **on** (**end**) clauses detect when the most inner closure is created or destroyed. This closure is created when all of the following three conditions become true:

1. The entry bound to the `entry` variable is an entry in the list bound to the `list` variable;

2. The `entry` has a successor in the `list`, which is determined by comparing its index to the size of the list (note that adding to the end of the list causes the previous last element to have a successor, and therefore this condition becomes true);

3. With respect to the comparator bound to the `comparator` variable, the `entry` is unsorted with respect to its successor, which is bound to the `next` variable.

The closure that is created when these three conditions become true causes the unsorted counter to be incremented by one. When any one of these three conditions becomes false, the closure is destroyed and the unsorted counter is decremented by one. The conditions become false when the entry is removed from the list, when it no longer has a successor, or when it becomes sorted with respect to its successor. The latter condition occurs when the entry's state changes, when the successor's state changes, or when the entry has a new successor. In a language like Java, each of these situations would often be coded individually, which results in code that is about five times the size of the code in Figure 2.18. The SuperGlue code is more concise because it is able to use signals to encapsulate these situations from glue code.

Note that the code in Figure 2.18 is very time efficient but not space efficient. List insertion and deletion has a time complexity of $O(1)$ because only the successors of elements adjacent to the insertion or deletion need to be recomputed. However, this time complexity comes at the expense of space complexity, which is $O(N)$ because event handlers are installed on every node of the list. If list element sorting order is immutable, space complexity can be reduced at the expense of time complexity by checking if a list is still sorted after every insertion and deletion. This tradeoff cannot easily be expressed in SuperGlue, which lacks the flexibility of Java code in expressing specialized continuous evaluation algorithms.

## 2.4 Class Implementations

Glue-code programmers do not often need to be concerned with object implementations because gluing objects together does not often require the creation of new kinds of objects. When objects are not implemented by the glue-code programmer, a one-level view of the program is sufficient and easier to understand. However, we provide class-definition mechanisms to glue code programmers because SuperGlue code can be reused in the form of classes. Additionally, SuperGlue library implementors need class-definition mechanisms to create class libraries.

Before describing how classes are defined in SuperGlue, it is useful to describe SuperGlue's code architecture. SuperGlue has a simple module system for organizing class declarations and definitions, where each module is defined in its own source code file. Besides containing class declarations and definitions, modules also contain glue code that connects the imports of these class definitions. These connections are considered defaults that can be overridden in glue code. For example, the module that declares a user-interface table class also connects the empty list to all rows of all user-interface tables. Connections defined in a module are not encapsulated, meaning they are a part of the module's public signature, and can be overridden in glue code according to the prioritization rules described in Section 2.2.5. A module must be imported into the namespace of another module or class definition before its classes can be referenced. Module import is analogous to package import in Java. However, module import is not only a namespace management mechanism, because default connections defined in a module are added to the circuits of class definitions that import the module.

Within a module, SuperGlue classes are implemented in one of two ways. First, a class can be implemented with SuperGlue code that glues objects together. Defining classes that contain SuperGlue code is useful for structuring and reusing glue code, and eliminates the need for other code-structuring abstractions in SuperGlue. Section 2.4.1 describes how classes are implemented in SuperGlue. Second, a class can be implemented in another programming language. This approach allows SuperGlue classes to reuse existing code that is not written in SuperGlue. Also, other programming languages can be better suited than SuperGlue for expressing classes whose implementations must

deal with low-level control flow issues or are performance sensitive. Section 2.4.2 describes how SuperGlue classes are implemented in other programming languages.

### 2.4.1 SuperGlue Implementations

The SuperGlue code of a class glues objects together. Within a class implementation, the object that is being implemented is referred to with the **`this`** keyword. Compared to an object that is being connected from its outside, the imports and exports of an object being connected from the inside switch roles: connections connect to the object's exported signals, while expressions can reference the object's imported signals.

As an example of how a class is implemented in SuperGlue, consider the declaration and implementation of the `DetectSorted` class shown in Figure 2.19. The `DetectSorted` class enables the reuse of the SuperGlue code that was originally shown in Figure 2.18. In this class, the list whose sorting status is being checked and the comparator used to compare entries are imported as signals, and if the sort status of the list is exported as a signal. The implementation of the `DetectSorted` class declares a `Counter` object that is created each time the `DetectSorted` class is instantiated into an object. The **`this`** keyword in the `DetectSorted` implementation represents the `DetectSorted` object that is being implemented. Otherwise, the implementation code in Figure 2.19 can be understood in the same way as the glue code in Figure 2.18.

Inner objects are created in the SuperGlue implementation of a class through a syntax that is similar to a connection query. When the target type of a connection query is specified to be an inner type that is accessed through **`this`**, the condition creates a new inner object that is connected from the target expression of the query. As an example, consider the following SuperGlue code:

```
if (entry = <this.Entry> this.list.get(i))
  if (entry.accept) ...
```

The above code creates an entry inner object for the purposes of accessing the inner object's `accept` import. This inner object is created by connecting the first element of the implemented object's `list` import. As demonstrated by this example, inner objects can be implemented as wrappers around other values, in which case they do not have

```
/* Signature of the DetectSorted class. */
class DetectSorted {
  import input      : List;
  import comparator : Comparator;
  export sorted : boolean;
} with {
  let unsorted_count = new Counter;
  this.sorted = (unsorted_count.result == 0);

  on (entry = this.input.all)
  if (entry.index < list.size - 1) {
    let next = list.get(entry.index + 1);
    if (this.comparator.compare(entry, next) > 0) {
      on (begin) do unsorted_count.increment;
      on (end)   do unsorted_count.decrement;
    }
  }
}
```

**Figure 2.19**. The declaration and implementation of the `DetectSorted` class.

their own identities or state. It is for this reason that inner objects can flexibly represent the ports of their containing object.

The SuperGlue implementation of a class is expressed as a closure definition without an enclosing lexical scope. The creation of an object with a SuperGlue implementation is similar to the creation of a closure: when the object is created, its constituent objects are initialized, **do** statements are performed, and **on** statements are activated in their lexical order.

SuperGlue does not need procedures and procedure calls to reuse connection code. Instead, classes in SuperGlue have most of the qualities of procedures: their declared imports are the same as procedure parameters, and objects instantiated from classes are the same as procedure calls. Although the lack of a procedure construct is extremely unusual in a programming language, the use of classes in place of procedures in SuperGlue makes the language smaller and potentially easier to use. However, unlike procedures, classes in SuperGlue cannot be recursively instantiated. As a result, SuperGlue does not support general recursion when building connection hierarchies. Instead, only con-

nection variables can be used to build connection hierarchies of unbounded sizes. The use of connection variables is less powerful than general recursion because connection variables provide only limited access to the connection graph.

### 2.4.2 Drivers

A SuperGlue class that is implemented in Java is referred to as a *driver*, which is a wrapper around a Java class. Drivers can extend other drivers in a way that mirrors the extension relationships of the classes being wrapped. For example, the `TableView` class extends the `Widget` class, and the Swing `JTable` class wrapped by the `TableView` class extends the Swing `JComponent` class wrapped by the `Widget` class.

The signals and streams of a driver are represented as Java driver objects. The Java interfaces implemented by signal driver objects are shown in Figure 2.20. The `SignalCircuit` interface represents the behavior of a signal with three methods: a `current()` method that is called to get the signal's current value, an `install()` method to install an observer object that observes changes in the signal's value, and an `uninstall()` method that uninstalls a previously installed observer object. Driver objects that represent imported signals and streams are provided by the run-time to the driver. Driver objects that represent exported signals and streams are provided by the driver to the run-time.

```
interface SignalCircuit {
 Value  current(Value targ, Value[] args);
 void   install(Value targ, Value[] args,
                            Object key, SignalObserver obs);
 void uninstall(Value targ, Value[] args,
                            Object key);
}
interface SignalObserver {
 void changed(Value oldValue, Value newValue);
}
```

**Figure 2.20**. The `SignalCircuit` and `SignalObserver` Java interfaces, which represent SuperGlue signals in Java code.

When a driver is instantiated, the driver is called to create a Java object that is then wrapped by the run-time in a SuperGlue object. After the SuperGlue object is created, the driver is called to initialize the SuperGlue object. SuperGlue object creation and initialize occur in different phases, where the signals of other objects can only be accessed by the driver during the SuperGlue object's initialization phase. As an example, the `Label` driver class's initialization code is shown in Figure 2.21. The `Label` driver is a wrapper around the `JLabel` class of the Swing user interface library. The initialization code in Figure 2.21 does two things. First, the initialization code sets the label's text to the current value of its imported `text` signal. The `circuit` method call on the value that represents the label object is used to access the circuit of the label object's imported `text` signal. The `current` method is called on this circuit to get the current value of the `text` signal, which can then be used as the label's text. Second, the initialize code installs an observer that updates the label whenever the value of the `text` signal changes. This observer is installed using the circuit's `install` method.

The `circuit` method of a class driver is called to access object and inner object exports of the class being being defined. As an example, consider the code for the `Thermometer` driver that is shown in Figure 2.22. The `Thermometer` class exports

```
class Label extends Driver {
  Object create() { return new JLabel(); }
  void init(ObjectValue object) {
    JLabel label = object.wrapped();
    SignalCircuit circuit = object.circuit(null, TEXT);
    text = circuit.current(object, null).toString();
    label.setText(text);
    text = circuit.install(object, null, label,
      new SignalObserver() {
        void changed(Value oldValue, Value newValue) {
          label.setText(newValue.toString());
        }
    });
  }
}
```

**Figure 2.21**. The implementation of the `Label` driver that is declared in Figure 2.2.

```
class Thermometer extends Driver {
  Object create() { return new Socket(); }
  void init(ObjectValue object) {
    /* initialize and connect socket */
  }
  Circuit circuit(InnerType ftype, Member mmbr) {
    if (ftype == null && mmbr == TEMPERATURE) {
      return TEMPERATURE_CIRCUIT;
    }
  }
  Circuit TEMPERATURE_CIRCUIT = new SignalCircuit() {
    Value current(Value targ, Value[] args) {
       int val = ((Socket) targ.wrapped()).read();
       return new IntegerValue(val);
    }
    void   install(Value targ, Value[] args, Object key,
          SignalObserver obs) { ... }
    void uninstall(Value targ, Value[] args, Object key) { ... }
  };
}
```

**Figure 2.22**. The implementation of the `Thermometer` driver class that is declared in Figure 2.2.

a `temperature` signal that is implemented by reading data through a socket. This implementation is expressed as a Java object that implements the `SignalCircuit` interface in Figure 2.20. The driver returns this objects whenever the circuit for the `Temperature` signal is requested.

Drivers usually wrap existing Java classes that are designed to work in an MVC architecture. Although wrapping such classes to work in SuperGlue is not trivial, it is usually possible to write class drivers with a reasonable amount of work, which we define as substantially less work than is needed to rewrite the Java classes being wrapped. Section 3.1 evaluates how much work it takes to adapt a Java class library to work in SuperGlue, and what kind of libraries can be reasonably adapted to work in SuperGlue.

The description in this section of how drivers are implemented is the approach of our current prototype. This approach has two drawbacks. First, driver implementations are very verbose with the code needed to locate and invoke signals. Although some convenience methods can help, this code is still very ugly and can be the source of many

errors. Second, our current driver model does not support compilation. Support for compilation in our implementation requires encoding additional information on how a signal can be implemented. We discuss performance issues in Section 4.5.

## 2.5 Discussion

SuperGlue was designed with the goal of making it as easy to use as possible. To this end, we have limited SuperGlue to a small set of orthogonal constructs, all of which have been demonstrated in this chapter. SuperGlue lacks many powerful abstractions such as procedures, recursion, or looping constructs because SuperGlue's programming model requires the hiding of control-flow details. These abstractions obscure continuous program behavior, interfere with SuperGlue's declarative semantics, and are often not needed to express glue code. To deal with the scalability issues of a basic port-connection paradigm, we introduced new constructs in the form of inner objects, connection variables, and connection queries. SuperGlue is not the first language to deal with these problems, and we describe alternative solutions in Chapter 5. Our solution differs from these other solutions because of our problem domain, which is the gluing together state-processing components.

Many of the problems we tackled in the design of SuperGlue do not have straightforward solutions. As a result, our chosen solutions involve making controversial tradeoffs. We do not claim that our design decisions are optimal, but the tradeoffs we make suit our domain. Additionally, SuperGlue's design is incomplete in many areas, such as error handling. The rest of this section discusses our most controversial design decisions and areas in the language whose design have not been explored.

### 2.5.1 Why Inner Objects?

SuperGlue's inner object abstraction is unique and is not very similar to any other abstraction in an existing programming language. Inner objects solve the connection scaling problem in an object-oriented way that does not sacrifice too much of the port-connection paradigm's simplicity. Alternative approaches either rely on functions [6] or combinators [20], or work through more graph-oriented pattern-matching mechanisms.

These approaches make different tradeoffs: functions are more powerful but obscure component dependencies, combinators preserve functional purity but are radically different from what Java programmers are used to, and graph-oriented pattern-matching mechanisms enable direct reasoning about the connection graph but requires building a concrete connection graph, which cannot be supported with existing Java components.

We specifically designed inner objects to facilitate the reuse of existing Java components that conform to MVC architectures, such as Swing user-interface [40] and JavaMail [39] components. These Java components often do not maintain the object graphs that would facilitate a more direct object representation of their members. Instead, drivers can create inner objects to represent logical public members of an object when they are exposed by the Java components. If SuperGlue code did not have to interface with existing Java components, we probably would have chosen an abstraction that is more conventional than the inner object abstraction. However, we have found that the flexibility that inner objects provide is useful even if we are building new components from scratch.

### 2.5.2 Universal Quantification vs. Iteration

Universal quantification is very different from iteration: universal quantification is used to describe properties over all values of some set, while iteration is used to access all values of some set. SuperGlue has separate abstractions to express universal quantification and iteration: connection variables, which are universally quantified over connections to objects or inner objects of specified types, and iterator streams, which are used to iterate over values. Connection variables can only be used to connect object ports: they cannot be used to iterate over the objects of a type. Iterator streams cannot be used to connect the ports of values being iterated over because the identities and circuit locations of these values cannot be determined. On the other hand, iterator streams can be used to iterate over a graph of objects and inner objects for a variety of purposes. However, they must be accessed from the root of the program's connection graph.

### 2.5.3 No Recursion or Loops

SuperGlue has no general-purpose recursion or looping abstractions because these abstractions would seriously complicate SuperGlue's semantics without significantly enhancing SuperGlue's purpose as a glue code language. Without recursion or loops, SuperGlue relies on iterator streams to iterate over sets of values and connection variables to abstract over port containers. These two abstractions can be used to encode most of the logic that is often encoded in glue code, while more complicated traversal logic should often be encoded in component implementations. For example, in Section 3.3, we describe how generic tree-traversal logic is encoded in component implementations and configured with inner objects.

### 2.5.4 Error Handling

SuperGlue does not currently have error-handling abstractions. As supported in existing object-oriented languages, exception handling does not make much sense in SuperGlue, which lacks a procedure activation stack. Presently, event streams can be used to communicate failures that occur at discrete points in time, but are not very elegant in communicating errors that occur on signals, which include built-in unconnected and ambiguous connection errors. We have yet to explore what it means for a signal access to fail, and what kind of recovery should be possible. Section 6.2 describes strategies for supporting error handling in SuperGlue.

### 2.5.5 Concurrency

SuperGlue's programming model naturally supports a form of concurrency that is based on events and does not involve multithreading. A component in SuperGlue can independently perform its computations but must communicate changes to other components by firing events that are dispatched in one thread. It has been argued that event-based concurrency is easier to use, is less error prone, and can perform better than multi-threaded concurrency on single processor machines [10]. For this reason, although Java provides strong support for multithreading, most of its core and input-output libraries are based on event-based concurrency. Our design of SuperGlue acknowledges this trend

by providing better support for events, with abstractions that hide event handling, while forgoing multithreading abstractions.

Communication between components in SuperGlue must currently occur in a single thread. Although component implementations can themselves be multithreaded, these threads cannot directly communicate with other components. SuperGlue's restriction on multithreaded communication is similar to the restrictions that most Java libraries place on multithreaded access. For example, Swing [40] user-interface components can only be accessed from a program's user-interface thread. Other threads can indirectly access user-interface components through the `invokeAndWait()` or `invokeLater()` methods in the `SwingUtilities` class. In our current implementation, SuperGlue code for user-interface programs always executes in a program's user-interface thread, and any observable state changes must be performed in the user-interface thread.

In user-interface programs, multithreaded concurrency is necessary to support long running operations, and so SuperGlue should support some form of multithreading. However, SuperGlue cannot easily be enhanced to support multithreading. Atomicity issues, which are described in Section 4.3, become very difficult to deal with when signal change events can be dispatched from multiple threads. Section 6.2 describes strategies for supporting multithreading in SuperGlue.

# CHAPTER 3

# EVALUATION

This chapter evaluates how SuperGlue improves the modularity of programs that reuse state-processing components. Our evaluation is divided into three case studies:

– The first case study in Section 3.1 explores how libraries of components are implemented in SuperGlue. SuperGlue's utility depends heavily on well-designed libraries of components. The main subject of this case study is a user-interface library that wraps Java's Swing user-interface library [40]. We show that this library can be designed in a straightforward way and implemented with a reasonable amount of code.

– The second case study in Section 3.2 compares how user-interface programs are implemented in SuperGlue and Java. The subject of this case study is an email client, which is representative of programs where users interact with state that changes continuously. We show that SuperGlue can reduce by half the amount of code needed to implement an email client.

– The third case study in Section 3.3 explores how easily batch-style programs can be recrafted in SuperGlue with state-processing features. The subjects of this case study are the parsing and type checking components of a compiler, which traditionally do not process state. We argue that the amount of code needed to reuse state-processing parsing and type checking components in SuperGlue is comparable to reusing versions of these components that do not process state.

The initial evaluation presented in this chapter is meant to show that SuperGlue's use in program development can be feasible and worthwhile. Although these case studies are

not very comprehensive, they do show how SuperGlue can improve the modularity of real programs in popular application areas.

## 3.1 Libraries

As with any other programming language, SuperGlue's utility depends heavily on the existence of well-designed libraries. Only when these libraries exist can we begin to explore how SuperGlue improves program development. Although designing a good library is challenging for any programming language, library design is especially challenging in SuperGlue because the library should take advantage of SuperGlue's unique abstractions. For this reason, we have come up with three design guidelines for SuperGlue libraries:

- A substantial amount of a library's functionality should be exposed as signals rather than streams. Signals are easier to use than streams because they hide control-flow details from glue code.

- To promote library interoperability, different SuperGlue libraries should refer to the same interfaces when describing similar user-defined abstractions.

- Because SuperGlue is an object-oriented language, the functionality of a SuperGlue library should be organized into a class hierarchy. Each class should implement as much program functionality as possible, which minimizes the number of objects that must be explicitly declared and directly connected in a program. The interface of a class should then be divided into inner objects that are collectively connected by type in glue code.

Although libraries that do not follow these guidelines are still usable in SuperGlue, they are not much easier to use than similar libraries in other programming languages.

Following the above three guidelines can be challenging when a SuperGlue library wraps a library that is implemented in another programming language. Because library design guidelines in other languages can differ from those in SuperGlue, wrapping such a library requires some amount of re-architecting to realize SuperGlue's benefits. In this

dissertation, we focus on the wrapping of libraries that are implemented in Java. Because we have designed SuperGlue's class system to be similar to Java's class system, a SuperGlue library can reuse the class hierarchy of the Java library it is wrapping. However, the following re-architecting processes still need to be performed when wrapping a Java library:

- Identify user-defined abstractions in the Java library that should be expressed as SuperGlue interfaces. This can be straightforward for user-defined abstractions that are already expressed as Java interfaces or abstract Java classes. However, there are cases where a useful SuperGlue interface does not correspond directly to a useful Java interface. In such cases, programmers can write extra code to adapt different Java abstractions so that they conform to a common SuperGlue interface.

- Identify classes in the Java library that should be SuperGlue classes. Many classes in a Java library play only supporting roles and so should not be classes in a SuperGlue library. Supporting Java classes should be represented by inner object types.

- If possible, Java classes that express small amounts of functionality should be aggregated into new larger SuperGlue classes that express common usages of the classes together. While aggregation sacrifices flexibility because only common uses are supported, it makes SuperGlue libraries easier to use.

- Identify groups of methods in a Java class that should be represented as a signal or stream in a SuperGlue class. Whenever possible, the types of these signals and streams should be based on existing interfaces

### 3.1.1 GlueUI

GlueUI is a user-interface library that wraps Java's Swing user-interface library The design of GlueUI demonstrates how libraries should be designed in SuperGlue. GlueUI's class hierarchy is shown in Figure 3.1. The core abstraction in the GlueUI library is the widget, which can display and receive information from a user. Widgets are instances

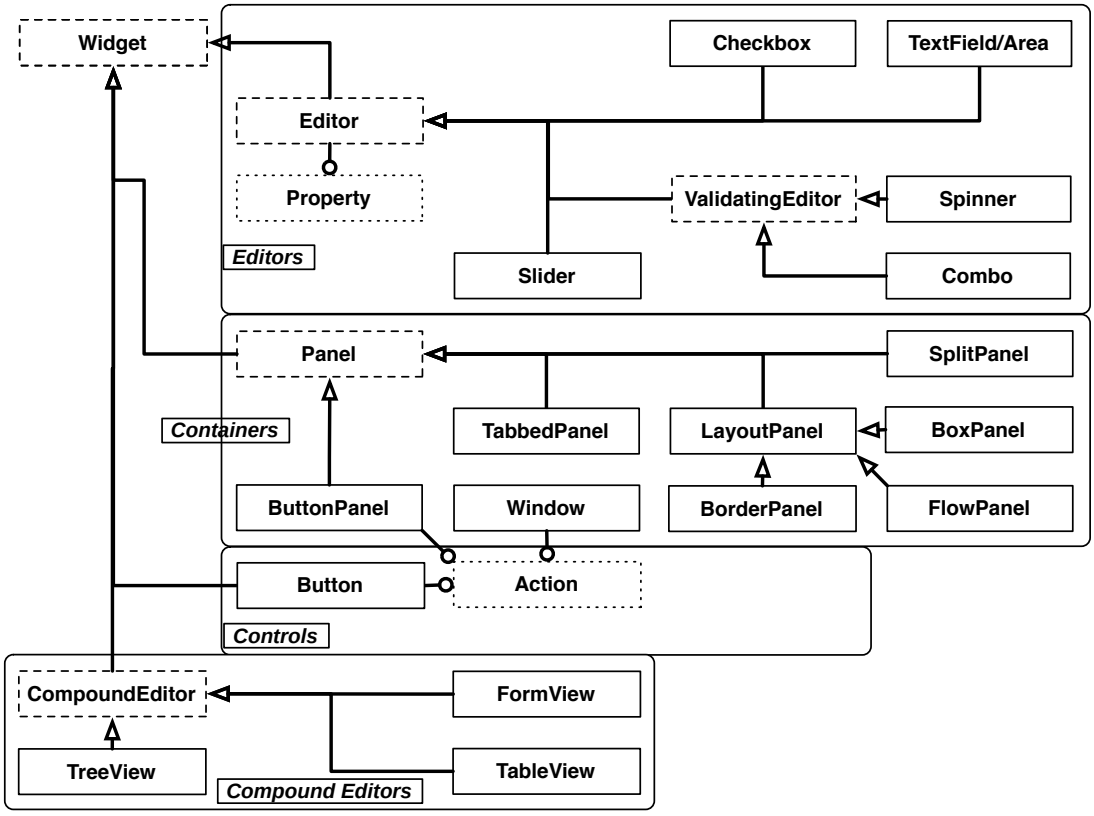


**Figure 3.1**. The class hierarchy of the GlueUI library: boxes are classes, dashed boxes are abstract classes; dotted boxes are interfaces, pointed arrows point to classes whose implementations are inherited, and circle arrows point to interfaces that are used in signal declarations.

of the GlueUI `Widget` class, which serves as a superclass for most of the classes in GlueUI. The `Widget` class wraps Swing's `JComponent` class, which is Swing's core widget abstraction. The rest of the classes in the GlueUI library fall into four categories:

– **Editors**, which view and edit elemental state known as *properties*;

– **Containers**, which contain other widgets;

– **Controls**, which perform command-like abstractions known as *actions* on user request; and

– **Compound editors**, which view and edit properties in large or unbounded size data structures.

This categorization of GlueUI classes is not exactly reflected in Swing's class hierarchy. Swing does not have explicit editor or compound editor classes, and in many cases expresses containers and layouts independently. These differences arise because of GlueUI's emphasis on ease of use differs from Swing's emphasis on flexibility.

### 3.1.2 Signal Identification

As much of a library's functionality as possible should be available through signals rather than streams. Although command and event streams can be used to wrap just about any Java method, they are harder to use because they expose glue code to control flow. Therefore, streams should only be used to describe functionality that is naturally imperative.

Signal identification in a Java library involves identifying multiple Java methods that express different aspects of the same continuous behavior. These methods can often be identified through naming conventions and design patterns [18], which is the case in Java's Swing and JavaMail libraries [39]. For example, the following methods in JavaMail's `Folder` class represent a signal that is a list of email messages:

- `getMessageCount();`
- `getMessage();`
- `addMessageCountListener();` and
- `removeMessageCountListener()`.

Each of these methods has `Message` in its name, which means they access email message state. The last two methods are involved in an observer design pattern, which is often used in a Java library to observe changes in state. Collectively, these methods can be used to implement a signal in a SuperGlue library that describes a list of email messages.

Once a Java library's conventions are understood, it can be straightforward to identify what methods in the library can be used to implement signals. Java methods in Swing are generally merged into one of three kinds of signals in GlueUI:

- Exported signals that represent continuous user input, such as the current edit value of a text field or the current tree node selection in a user-interface tree;
- Imported signals that represent widget configuration options, such as a widget's font and foreground color;
- Exported signals that represent modifiable widget properties, such as whether a window is currently visible.

The last kind of signal is a *property*, which is a signal that is augmented with a `set` command stream. A property in SuperGlue is expressed with the `Property` interface, which is declared in Figure 3.2, with the `get` signal to get and the `set` command stream to set the property's current value. Because properties are common in SuperGlue libraries, the `Property` interface is a part of SuperGlue's core library.

Distinguishing between configuration options and properties in Swing is tricky because they are expressed by similar kinds of methods. For example, a window's visibility is a property that is manipulated through the `getVisible()` and `setVisible()` methods, whereas the a widget's foreground color is a configuration option that is manipulated through the `getForeground()` and `setForeground()` methods. In a SuperGlue library, properties and configuration options are best expressed with different abstractions. A configuration option is easier to use than a property because it can be configured declaratively by connecting one imported signal, while a property must be set at some specific point in time. On the other hand, the value of a configuration option cannot be modified inside its containing object. For this reason, widget foreground color is a configuration option because widgets never internally change their foreground

```
interface Property<T> {
  port get : T;
  port command set(value : T) : void;
}
```

**Figure 3.2**. The declaration of the `Property` interface.

color, while window visibility is a property because windows can internally change their visibility in response to user input.

### 3.1.3 Interface Identification

When similar groups of signals and streams are declared in different classes, these signals and streams should be expressed in a shared interface. Expressing these groups of signals and streams as interfaces promotes interoperability by reducing the number of connections needed to glue objects together. Additionally, because inner types cannot be used outside of their declaring class, interfaces are the only way to express similarities between inner type declarations in different classes. Interfaces defined in SuperGlue's core library include the `List` interface, which is used to define list data, and the `Property` interface, which is declared in Figure 3.2. Interfaces can also be defined in noncore libraries. For example, the GlueUI library includes the `Action` interface, which is used to express user commands. Although noncore interfaces are unlikely to be used in unrelated libraries, they can be referred to in reusable connections that describe how objects in unrelated libraries are glued together.

As an example of how interfaces are used, consider how the `Property` interface from Figure 3.2 is used to simplify the use of a GlueUI editor. The abstract `Editor` class, which is declared in Figure 3.3, imports an `edited` property, which represents the state that is edited and viewed by an editor. The concrete `TextField`, `Checkbox`, and `Slider` classes can then be used to create editors that respectively view and edit string, boolean, and integer properties.

```
abstract class Editor<T> {
  import edited : Property<T>;
  ...;
}
class TextField  extends Editor<String> {...}
class Checkbox   extends Editor<boolean> {}
class Slider     extends Editor<boolean> {...}
```

**Figure 3.3**. The declarations of the `Editor`, `TextField`, `Checkbox`, and `Slider` classes.

When another library exports properties through the `Property` interface, GlueUI editors can be used to view and edit these properties directly. For example, a SuperGlue email library can export email message properties with the `Property` interface, which occurs in Figure 3.4. Given an email message bound to the `message` variable, the following SuperGlue code can be used to create an editor that views and edits the message's subject property:

```
let field = new TextField;
field.edited = message.subject;
```

As with any other programming language, when designing a SuperGlue library, it is important to identify new interfaces that are reused multiples times in this library. As an example, user interfaces must often deal with labels in many different contexts:

- Editor fields in a user-interface form are labeled according to the properties being edited.
- Columns in a user-interface table view are labeled according to the properties being viewed and edited in that column.
- User-interface actions are labeled according to what command they perform.
- Menus are labeled according to what actions they organize.

```
class Mailbox {
  class Message {
    export from     : Property<String>;
    export to       : Property<String>;
    export subject  : Property<String>;
    export received : Property<Date>;
    export sent     : Property<Date>;
    export deleted  : Property<boolean>;
    export flagged  : Property<boolean>;
    ...
  } ...
}
```

**Figure 3.4**. The declaration of the `Message` class nested in the `Mailbox` class.

To accommodate each labeling context, a `Labeled` interface is defined in the GlueUI library as follows:

```
interface Labeled {
  port text : String;
  port icon : Icon;
}
```

Although this interface is small, it is very useful. Because connection variables can be typed using interfaces, a connection can specify how objects created from classes in other libraries are labeled in GlueUI. As an example, the following connection expresses how an email message subject property is labeled in a user interface:

```
var (lbl : Labeled)
  if (lbl == mail.MESSAGE_SUBJECT)
    lbl.text = "Subject";
```

This glue code can label a message subject property whenever labeling is required in a user interface. In an email client, message subject properties are displayed as columns in table views, and as fields in form views. In both table views and form views, labels are computed through the `Labeled` interface. As a result, this glue code can be used to derive the column label and field label required when the message subject property is used in table views and form views.

A SuperGlue library can reuse interface abstractions that are present in the Java libraries they are wrapping. For example, Swing supports a user-interface action abstraction as a Java interface that represents button behavior, menu bar behavior, toolbar behavior, and so on. A similar action abstraction is supported in GlueUI as the `Action` interface, which is declared as follows:

```
interface Action extends Labeled {
  port enabled : boolean;
  port tooltip : String;
  port command action : void;
}
```

Besides declaring a command that is performed when the action is triggered, the GlueUI `Action` interface declares signal that configure when the action is enabled, how it is labeled, and what tool tip text can describe the action in a user interface.

Given SuperGlue's emphasis on ease of programming, interfaces should be used more aggressively in SuperGlue than they are in other object-oriented languages such as Java. Java libraries often express user-defined abstractions through conventions rather than interfaces. The benefit of the convention approach is that the user-defined abstraction allows more efficient use of functionality than is possible through a generic interface. For example, libraries often express lists through naming conventions rather than through Java's core `List` interface because different lists are more efficiently used in slightly different ways. The drawback of this approach is that the libraries are harder to use, which is why we emphasize interface identification in SuperGlue libraries.

### 3.1.4 Class Identification

As in most other languages, gluing together a small number of objects with coarse-grained functionality is easier in SuperGlue than gluing together a large number of objects with fine-grained functionality. Additionally, coarse-grained objects are more desirable in SuperGlue for two reasons:

- It is very difficult to create a large or unbounded number of objects because SuperGlue's connection-based programming model does not support loops or recursion.
- SuperGlue's support for inner objects, connection variables, connection queries, and interface abstraction increases the configurability of objects with functionality that is coarse-grained. As a result, coarse-grained objects can be used flexibly in programs.

As an example, GlueUI provides classes to create top-level label, editor, and button objects that can be used to create forms that view and edit lists of properties. With these objects, forms with a large number of edited properties are difficult to implement because a large or unbounded number of top-level editor objects, label objects, and button objects must be connected explicitly in glue code. For this reason, GlueUI provides the

`FormView` class, which is declared in Figure 3.5. Form view functionality is coarser-grained than label, editor, and button objects, whose functionality are aggregated inside form views. Although form views have coarse-grained functionality, they can still be used flexibly in user interfaces because they are configurable in the following ways:

- Edited properties are imported into a form view as a list of nested field objects. A field imports the signals of shared interfaces to obtain the field's edited property, a label for this property, and a configured editor object for this property. A field also imports a signal that controls when it can be edited.
- Form views import subject values, which allow glue code to configure what value's properties can be viewed and edited.
- Form views export save and revert command streams, which allow the form's functionality to be integrated with other GlueUI objects.

As an example of how form views are used, the glue code in Figure 3.6 creates an email composition window that views and edits in a form the to (receiver), subject, and body properties of a newly created email message. A toolbar is used to provide access to email composition actions, such as the send action. The send action is a composite that saves the form's edits to the composed email message, sends the email message, and closes the email composition window.

```
class FormView extends CompoundEditor {
  inner Field imports PropertyID, Labeled, FindEditor {
    import editable : boolean;
  }
  import fields   : List<Field>;
  import subject  : PropertyContainer;
  import editable : boolean;
  export save     : Action;
  export revert   : Action;
  ...
}
```

**Figure 3.5**. The declaration of the GlueUI `FormView` class.

```
let window = new Window;
let toolbar = new ToolBar;
let form    = new FormView;
let send    = new CompositeAction;
let panel   = new BorderLayout;

panel.center = form;
panel.north  = toolbar;
window.display  = panel;

form.fields = [messages.to,
               messages.subject, messages.body];
form   .subject  = new_message;
toolbar.subjects = [send, ...];

send.text = "Send";
on (send.activated) {
  do form.save;
  do new_message.send;
  do window.visible.set(false);
}
```

**Figure 3.6**. Glue code that is used to compose and send an email message.

Form views are compound editors that aggregate the viewing and editing of multiple properties. Besides form views, other compound editors in GlueUI include tree views and table views, which were used in the examples in Chapter 2. Compound editors adhere to SuperGlue's library design guidelines more strongly than other GlueUI classes. As a result, a program that can extensively use compound editors is easier to write in SuperGlue than in other languages.

### 3.1.5 Comparison

We have so far shown through examples that SuperGlue libraries can be significantly easier to use than Java libraries with similar functionality. Here we describe how SuperGlue libraries can be easier to use than Java libraries by comparing libraries based on their feature sets and number of declarations. In Section 3.2, we compare how a specific program is implemented in SuperGlue and Java.

Our comparison involves GlueUI and Swing. Because GlueUI classes represent widgets, we compare how GlueUI widgets compare to Swing classes. The results of this comparison are as follows:

- **Widget**: The base GlueUI widget class has three signals that correspond to three methods in the Swing class `JComponent`. The functionality of most methods in `JComponent` are not made available in GlueUI's `Widget` class, which sacrifices some flexibility but makes GlueUI easier to use.

- **Editors**: A base GlueUI editor has seven signals. Editing is not well-defined in Swing, i.e., there is no common editor base class in Swing. As a result, GlueUI editors are substantially different from the `JCheckBox`, `JTextField`, `JSlider`, and `JSpinner` Swing classes. For property editing tasks, GlueUI editors are easier to use than these Swing classes because they use abstractions that specifically support editing. On the other hand, GlueUI editors cannot be very easily used in the nonediting contexts that these Swing classes can be used in.

- **Controls**: Both GlueUI controls and Swing control classes are parameterized by action interfaces. As a result, using controls in GlueUI and Swing is very similar.

- **Containers**: For ease-of-programming, panel and layout functionality are expressed in combined GlueUI classes. Menu bar functionality is also combined with window functionality into a single `Window` class. Although these combinations are possible in Java, Java's object model makes them awkward to express so these functionalities are separated in Swing. SuperGlue's support for inner objects makes these functionalities convenient to combine in GlueUI.

- **Compound Editors**: Swing does not support a form view class. A GlueUI tree view consists of 11 signals, while Swing's `JTree` class requires the use of about 30 methods and 4 supporting classes to use similar features. A GlueUI table view consists of 10 signals, while Swing's `JTable` class requires the use of 25 methods and 3 supporting classes to use similar features. For reasons mentioned in Sec-

tion 3.1.4, GlueUI's compound editors are much simpler than Swing's compound editors.

### 3.1.6 Library Implementations

The driver implementation of a SuperGlue library must translate the abstractions of the wrapped Java library into effective SuperGlue abstractions. This translation involves using Java methods to implement SuperGlue signals, streams, and nested objects. In many cases, drivers must translate library abstractions so they can be represented by interfaces such as lists, which are shared with other SuperGlue libraries. In GlueUI, examples of these kinds of translations occur in the following contexts:

- All Swing widget classes are wrapped so that their properties are expressed with the `Property` interface.
- Each editor-like widget in Swing is wrapped in a driver so that its target state is expressed with the `Property` interface. Each editor is also wrapped so that its current edited value is expressed as a signal.
- Swing's `JTree` class is wrapped in a driver so that its lists of selected and expanded nodes are expressed as signals of the `List` interface. In both the selection and expansion cases, extra list data structures must be maintained because selected and expanded nodes are not accessible as lists.
- Swing's `JTable` class is wrapped in a driver so that its lists of selected rows and columns are expressed as signals of the `List` interface. As in the `JTree` class, extra list data structures must be maintained because row and column selection is not accessible as lists.

While far from trivial, our wrapping of Swing classes in GlueUI is manageable because signal-like behavior can be obtained with the observer design pattern, which is used throughout Swing's classes. The number of lines of Java code required for each GlueUI class are listed in Figure 3.7. Most drivers in GlueUI require less then 100 lines of Java code. The `TreeView` and `TableView` classes have the most complicated drivers in

| *Class File* | *Line Count* |
|---:|:---:|
| **Widget** | 61 |
| **Label** | 44 |
| **Editor** | 41 |
| **TextWidget** | 84 |
| **TextArea** | 39 |
| **TextField** | 44 |
| **Slider** | 147 |
| **Checkbox** | 82 |
| **Control** | 11 |
| **Button** | 81 |
| **Panel** | 58 |
| **LayoutPanel** | 11 |
| **FlowPanel** | 64 |
| **SplitPanel** | 85 |
| **BoxPanel** | 71 |
| **BorderPanel** | 37 |
| **Window** | 201 |
| **CompoundEditor** | 89 |
| **TreeView** | 444 |
| **TableView** | 611 |
| **Form** | 455 |
| *Total* | 2910 |

**Figure 3.7**. A listing of GlueUI Java driver class files.

GlueUI, which require 444 lines of code and 611 lines of Java code, respectively. These drivers are more complicated than other drivers in GlueUI because these two classes have a larger number of signals, and the drivers must adapt many abstractions into common SuperGlue interfaces. These larger drivers are needed because the `TreeView` and `TableView` classes realize SuperGlue's modularity benefits better than other GlueUI classes. The driver of the `Form` class requires 455 lines of Java code because forms must be implemented from scratch. In total, GlueUI drivers for 21 classes require 2,910 lines of Java code for the feature set we have implemented, which is sufficient

to express the email client program described in Section 3.2. As a point of comparison, Swing's implementation for the features implemented in GlueUI includes 31,000 lines of code in the `javax.swing` package, 9,500 lines of code in the `javax.swing.tree` package, and 4,700 lines of code in the `javax.swing.table` package. This code excludes code that implements the `Form` class as well as platform-specific code that is located outside of the `javax.swing` packages.

## 3.2 User-interface Programs

This section presents a case study of how SuperGlue can be used in the construction of a program with a user interface. This case study shows how SuperGlue is useful in user-interface programs where users view and change volatile data, which includes programs in many kinds of domains. An email client was also chosen specifically for our case study because email clients are well understood and widely used.

This case study compares the SuperGlue and Java implementations of an email client. The SuperGlue email client implementation uses SuperGlue's core library, the GlueUI library described in Section 3.1, and a single class SuperGlue library that wraps classes in Java's JavaMail library. A screen shot of this email client is shown in Figure 3.8. The Java email client implementation uses Java's core libraries, the Swing user-interface library, and the JavaMail library. Our comparison is organized according to how much code is needed to express the following email client features:

- **Navigation**, which allows a user to navigate mailboxes, folders, and messages. Navigation is divided into three views: a folder-view tree, which views the folders of installed mailboxes, a message-view table, which views rows of message headers, and a content-view form, which views the contents of a message. In Figure 3.8, the folder view is in the upper left-hand corner, the message view is in the upper right-hand corner, and the content view is in the bottom portion of the screen shot.

- **Deletion**, which allows a user to delete email messages. Deleted messages are highlighted in the message view table, and the user can expunge deleted messages in a folder that is selected in the tree view.

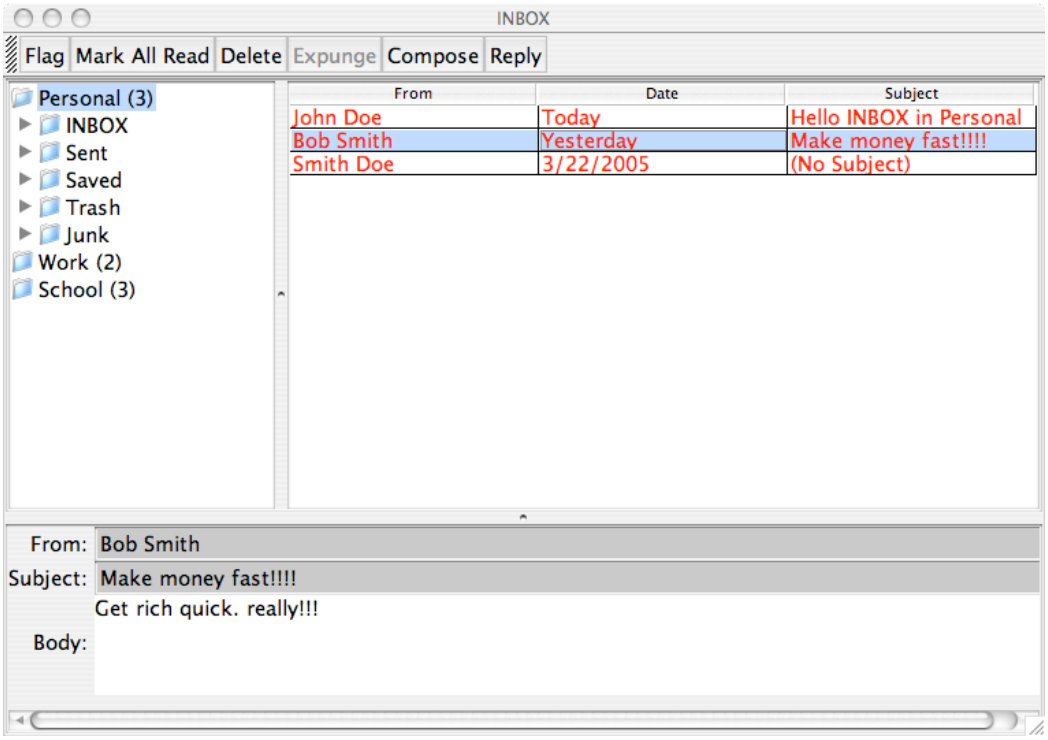


**Figure 3.8**. A screen shot of an email client that is implemented in SuperGlue.

– **Composition**, which allows a user to compose and send a new message, and reply to an existing message.

The methodology used in our comparison involves measuring two metrics in each implementation: lines of code and number of operations. While line counts are accurate metrics in measuring verbosity, they are not very accurate metrics in measuring complexity. Although verbosity and complexity are loosely related, code that is more verbose can aid in readability and is not necessarily more complicated. For this reason, we also measure the number of operations needed to implement a feature. We count only operations that are defined by libraries and not built into the programming language. We do not count type declarations, local variable assignments, control flow constructs, and so on, which contribute to verbosity but do not make a library more difficult to use. For example, a

method call in Java or connection in SuperGlue are both operations that we count, while variable uses in Java and SuperGlue are operations that we do not count. Because the operations we count are related to using a library, they are a more accurate measure of complexity than line count.

The following SuperGlue operations are counted in our comparison:

- A creation of an object is one operation.
- A connection to a signal or stream is one operation.
- A signal or stream access is one operation plus the number of arguments used in the access. Accessing a signal through multiple levels of signals does not add to the number of operations because hierarchical signal organizations in SuperGlue (or, as we mention later, hierarchical object organizations in Java) are good designs that do not increase complexity. For example, both **`table.`**`selected_rows` and **`table.`**`rows.selected` count as one operation.
- A connection query is one operation.
- A class declaration counts as one operation. Each import and export of the class is another operation.
- Declaring a connection variable is one operation. We count connection variables as operations because they are used in place of higher-order functions and callbacks in SuperGlue.
- Variable uses, arithmetic operations, and constant uses do not count as operations.

As an example, consider counting the number of operations in the following SuperGlue code:

```
val folderView = new TreeView;
var (node : folderView.Node) {
  if (folder = <mail.Folder> node) {
    node.children = folder.sub_folders;
    node.text = folder.name +
      " (" + folder.unread + ")";
}
```

This SuperGlue code contains eight operations: one from an object creation, one from the declaration of a dispatch variable, one from one connection query, two from two signal connections, and three from three signal accesses.

The following Java operations are counted in our comparison:

- An creation of an object is one operation plus one operation for each constructor argument used in the creation.
- A method call, array access, or length access is one operation plus one operation for each argument used in the call. As in SuperGlue, calling a method through multiple levels of calls does not add to the number of operations. For example, the following calls both count as one operation: `table.isRowSelected()` and `table.getRow().isSelected()`. We do not need to use temporary variables in the Java code we are comparing against, so we do not encounter cases like `row = table.getRow()` followed by `row.isSelected()`.
- A class implementation is one operation. A method or constructor implementation is one operation plus one operation for each of their formal parameters, excluding **this**. Each field of a class is one operation.
- An **instanceof** test counts as one operation. A downcast does not count as an operation because it does not involve any program logic beyond error detection.
- Local variable declarations, field uses, local variable uses, control flow construct uses, arithmetic operations, and constant uses do not count as operations.

As an example, consider counting the number of operations in the following Java code:

```
folderView = new JTree();
folderView.setModel(new TreeModel() {
  int getChildCount(Object node) {
    if (node instanceof Folder)
      return ((Folder) node).list().length;
  }
  Object getChild(Object node, int index) {
    if (node instanceof Folder)
      return ((Folder) node).list()[index];
```

```
  }
});
```

This Java code contains twelve operations: two for creating two objects, one for calling one method, five for implementing two methods with three arguments, two for two **`instanceof`** tests, one for one array length access, and one for one array element access.

### 3.2.1 Results

The results of our comparison are shown in Figure 3.9. By far the largest reduction in program complexity is obtained in the SuperGlue implementation of the navigation feature. This reduction is large for two reasons. First, the navigation feature uses compound editors, which are objects with a large amount of functionality and signals. Second, the navigation feature involves a lot of continuous behavior that can be expressed more concisely in SuperGlue than in Java. Besides the navigation feature, other features involve only a minor amount of continuous behavior, and so their implementations do not benefit as much from SuperGlue as the navigation feature's implementation does.

When comparing operations, the composition feature does not benefit from being implemented in SuperGlue. This is because the composition feature does not involve very much continuous behavior: a user only hits the compose button, fills out a form, and hits a send button. Only the parts of a program that involve a significant amount of state-processing benefit from being implemented in SuperGlue rather than Java. The composition feature's implementation is not harmed by being implemented in Super-

| | **Line Counts** | | | **Operations** | | |
|---|---|---|---|---|---|---|
| *Features* | *Java* | *SuperGlue* | $\frac{Java}{SuperGlue}$ | *Java* | *SuperGlue* | $\frac{Java}{SuperGlue}$ |
| **Navigation** | 147 | 51 | 2.8 | 265 | 110 | 2.4 |
| **Deletion** | 24 | 23 | 1.0 | 45 | 35 | 1.3 |
| **Composition** | 54 | 43 | 1.3 | 96 | 76 | 1.3 |
| **Total** | 225 | 117 | 1.9 | 406 | 221 | 1.8 |

**Figure 3.9**. A comparison of email client features as they are implemented in SuperGlue and Java.

Glue, which is due to effectiveness of streams in expressing imperative and discrete behavior.

Only the implementation of the navigation feature achieves a significant reduction in line count and operations when implemented in SuperGlue. Because the navigation feature is by far the largest feature in the Java program, the overall reduction in line count and operations when implementing our example email client in SuperGlue is around two. As a result, the use of SuperGlue in building a program can be worthwhile when only some of the program's functionality involves continuous behavior.

The delete and composition features do not involve very much continuous behavior: message deletion and folder expunging depend on pushing buttons, while message composition depends on popping up a form and sending a message on a user's command. Both of these behaviors are heavily imperative and therefore programmers do not benefit very much from implementing the features in SuperGlue. Given a user interface program that depends heavily on imperative user input and output, such as web-based programs, SuperGlue's use is of little benefit to programmers. Only when a user-interface program has a significant number of continuous user-interface features can SuperGlue be useful in its construction.

## 3.3 State-processing Programs

The email client case study in Section 3.2 shows how SuperGlue can reduce the code needed to implement existing kinds of applications that are assembled from state-processing components. The case study in this section shows how SuperGlue can be used to easily recraft existing batch-style applications that are not assembled state-processing components into interactive applications that are assembled out of state-processing components. With signals, the amount of code needed to assemble a state-processing component in SuperGlue is comparable to the amount of code needed to assemble a component with similar functionality that does not process state. For example, reusing a state-processing parser component in SuperGlue involves an amount of code that is comparable to reusing a parser that does not process state.

The subject of this case study is a compiler that is assembled out of state-processing components. Such compilers are used to implement language-aware editors, which provide programmers with immediate feedback about the syntactic and type correctness of their code. Language-aware editors arose out of research in syntax-directed editing [42], which advocated editing code as syntax trees rather than text buffers to reduce programming errors. Unlike syntax-directed editing, language-aware editing does not overly restrict programmers to expressing syntax tree modifications. Instead, knowledge about the language's syntax and semantics is used to continuously examine code as it is modified by a programmer. This continuous examination is used to detect errors, compute code completion advice, and enable language-aware search and replace.

The availability of language-aware editors is often limited to popular programming languages because they are difficult to implement. While the lexers, parsers, and type checkers of a standard compiler can be implemented as unidirectional data transformers, these same components in a language-aware editor must process state to continuously react to changes in the code being edited. Using state-processing compiler components in a language like Java is very difficult because glue code must deal with many state consistency details. SuperGlue's support for signals and objects can be used to improve language-aware editor implementations in the following ways:

- Source code token lists, parse trees, and symbol tables can be expressed as signals, which concisely represent how these stateful entities can change during editing.
- The transformation phases of a compiler, such as lexing, parsing, and type checking, can be expressed as rules, which enable flexible modularization of these tasks in glue code.
- SuperGlue's object-oriented connection model simplifies the configuration of continuous tree traversal components. This eases the implementation of type checkers and other kinds of static analyses.

The case study in this section describes how a language-aware code editor for some language can be implemented with SuperGlue libraries that were described in Section 3.1.

In this case study, we focus on the assembly of components that implement continuous parsing (Section 3.3.1) and type checking (Section 3.3.2). We show how reusing these components requires code that is comparable to reusing versions of these components that only transform data once. Section 3.3.3 discusses how the SuperGlue implementation of a language-aware editor differs from existing kinds of language-aware editors.

### 3.3.1 Parsing

Figure 3.10 shows the declaration of the `Parser` class, which is used to implement continuous top-down parsers. These parsers are handcrafted, meaning the lookahead tokens that disambiguate productions must be explicitly specified in glue code. The drawback of this approach is that the parser is more difficult to write than if a parser generator language, such as YACC, were used. The benefit of this approach is that parser expression can use the full power of the SuperGlue language, e.g., to implement expressive error handling. The `Parser` class eases continuous parser implementations in the following ways:

- The token input of the parser is expressed as an imported list signal. As a result, the list of tokens being parsed can change continuously.

- Parsing is rule-driven, as in popular domain-specific parser construction languages such as ANTLR [22]. Connection variables are used to express conditional connections to all parse tree nodes in a parser object.

- Parsing is configured by conditionally connecting the imported signals of a parse tree node. These imports specify the actual production type of the parse tree node, and the expected production types of its children.

- The expected production type of a parse tree node is connected to the parse tree node. As a result, glue code can determine the expected production type of a parse tree node through a connection query, and expected production types are used to prioritize connections to parse tree node in circuits.

```
class Parser {
  inner Token {
    import text : String;
  }
  import tokens : List<Token>;

  inner Input {
    import children  : List<Input>;
    export lookahead : Token;
    import result    : Node;
  }
  import input : Input;

  inner Node {
    export lookahead : Token;
    export children : List<Node>;
  }
  export root : Node;

  inner Error extends Input {}

  export errors  : List<Error>;
}
```

**Figure 3.10**. The declarations of the `Parser` class, which is used to implement continuous paring.

– A token lookahead list is associated with each parse tree node, rather than only with the parser object, because parse tree nodes can be processed simultaneously. The token lookahead list of a parse tree node is expressed as an exported signal, which can be used to guard connections to the imported signals of the parse tree node. As a result, how a parse tree node is parsed, which is determined by its imported signal connections, can change continuously as the list of tokens changes.

– Parse errors are indicated by signals in a parse tree node that are either unconnected or ambiguously connected. An error is an inner object whose signals can be connected to implement error recovery. Errors are also exported from the parser object as a list signal, which can be used to annotate errors in a text editor. When

an error is fixed through text editing, it is automatically removed from the exported error list.

– Different kinds of inner objects can be used to build a parse tree (`Input` objects) and to traverse the built parse tree (`Node` objects). For example, child nodes are imported during parse tree construction and exported during parse tree traversal. This separation also prevents cycles from accidentally being created between imported and exported signals.

Figure 3.11 shows an example of how a continuous parser is implemented with the `Parser` class in Figure 3.10. This code expresses rules to parse SuperGlue statements. The `input` connection variable is bound to a parser `Input` inner object that represents a position in the editor's token stream. The token at this position is accessible through the `Input` inner object's `lookahead` signal, which is used to compute the node result and children for the position. In the code of Figure 3.11, different parsing actions are taken if the `Input` inner object's lookahead token is bound to the `"let"` string or the `"var"` string.

The code in Figure 3.11 has a structure that is similar to the structure of hand-coded parsers that are assembled out of nonstate processing components. This provides evidence to our claim that the amount of code needed to use state-processing components in SuperGlue is similar to the amount code needed to use components with similar functionality but do not process state. In this case, configuring a state-processing parser component in SuperGlue is similar to configuring a normal parser component, which would look similar to the code in Figure 3.11.

### 3.3.2 Type Checking

Type checking involves one or more traversals of a parse tree to populate a symbol table and to compute if parse tree nodes are well typed. Type checking over a parse tree that changes continuously must also occur continuously. As a result, a traditional noncontinuous tree traversal algorithm, which visits nodes once in some specified order, cannot be used to implement type checking. Instead, a continuous tree traversal algo-

```
let lexer = new Lexer;
let parser = new Parser;

lexer.buffer = editor.buffer;
parser.tokens = lexer.tokens;

parser.input = STATEMENTS;
var (input : parser.Input) {
  if (input == STATEMENT) {
    if (input.lookahead == "let") {
      node.result = L_VAR;
      node.children = ["let", ID, '=', EXPR, ';'];
    }
    if (input.lookahead == "var") {
      node.result = C_VAR;
      node.children =
        ["var", '(', ID, ':', VTYPE, ')', STATEMENT];
    }
     ...
  }
  ...
}
```

**Figure 3.11**. Part of a parser implemented by configuring a `Parser` object.

rithm is expressed as a transformation of a tree into a single value or a list values. How this value or list is constructed depends on the functionality of the traversal.

The `TreeToSet` class, which is declared in Figure 3.12, implements a tree traversal algorithm that is generic with respect to the kind of tree being traversed and the kind of set produced to represent the result of a traversal. This class eases the implementations of continuous tree traversals in the following ways:

- The tree being traversed is expressed as a set of imported signals, meaning its changes are automatically propagated to the `TreeToSet` object.
- Traversal nodes are the inner objects of a `TreeToSet` object. Connections to traversal nodes configure what nodes are traversed in the traversal. The expression of a traversal is rule driven: connections to traversal nodes are specified through connection variables.

```
class TreeToSet {
  inner Result {}
  inner Node {
    import children : List<Node>;
    import result   : Result;
  }
  import root    : Node;
  export results : Set<Result>;
}
```

**Figure 3.12**. The declaration of the `TreeToSet` class, which is used to implement continuous tree traversals.

- The tree node being traversed is connected to traversal node inner objects. As a result, traversed nodes can be accessed in glue code through connection queries, which can guard connections to traversal nodes.

- The result of a traversal is a set of result inner objects that are connected to the `result` imports of all traversal node inner objects computed from the tree being traversed. As the tree changes, different values can be connected to `result` imports, and the set of result inner objects changes automatically.

Figure 3.13 shows an example of how a continuous undefined-symbol type checker is implemented using the `TreeToSet` class. This code creates two `TreeToSet` objects: the symbol table (**`symtab`**) object to track what variables have been defined, and the **`undefined`** object to track uses of variables that are not defined in the symbol table. Both `TreeToSet` objects traverse the parse tree. The symbol table object creates a set of variable names that have been defined by putting variable names declared by **`L_VAR`** and **`C_VAR`** statements into the symbol table's set. The undefined object then checks the token used in each variable use (**`VAR_USE`** expressions) is in the symbol table set. If the variable use is not in the symbol table set, the use is added to the undefined set, which causes the token to be colored red in the editor.

The code in Figure 3.13 does not consider position of a variable use with respect to declaration ordering and scoping. Such behavior cannot be obtained by depending on traversal order, which is not fixed in a continuous traversal. Instead, we must annotate the

```
let symtab = new TreeToSet;
symtab.root = parser.root;

var (node : symtab.Node) {
  if (node == L_VAR)
    node.result = node.children.get(1);
  else if (node == C_VAR)
    node.result = node.children.get(2);
  else node.result = null;
}

let undefined = new TreeToSet;
undefined.root = parser.root;

var (node : undefined.Node)
  if (node == VAR_USE &&
      !symtab.contains(node.children.get(0))) {
    node.result = node.children.get(0);
  } else node.result = null;

var (token : lexer.Token)
  if (undefined.results.contains(token))
    token.color = colors.red;
  else token.color = colors.black;
```

**Figure 3.13**. Code that implements continuous undefined-symbol type checking.

symbol table with positional information, which is checked when considering whether a variable use is undefined. Implementing and using these annotations only requires extra bookkeeping and does not fundamentally change the structure of the code in Figure 3.13.

As with continuous parsing, continuous type checking automatically changes its outputs, which in Figure 3.13 is the `set` signal of the **`undefined`** object, according to changes in the parse tree. As a buffer is edited and the parse tree changes, the set of declared and used variables can change, causing tokens to be added or removed from the undefined set. As a token moves into or out of the undefined set, its color changes immediately to reflect its new status, which provides immediate feedback to programmers.

Although our implementation of a continuous traversal is different from a conventional ordered tree traversal, the encoding of these two kinds of traversals are comparable in their code size. The traversal described in Figure 3.13 is similar in structure to the visitor pattern, which is commonly used to implement type checking with nonstate processing components.

### 3.3.3 Discussion

Unlike the case study in Section 3.2, we do not quantitatively compare the SuperGlue implementation of an language-aware editor to implementation in other languages for the following two reasons. First, we are unaware of any general Java libraries for implementing language-aware editors with continuous parsing features. Although Eclipse [21] provides libraries for implementing language-aware editors, and provides a Java editor that supports something like continuous parsing, there is no special support for implementing a continuous parser in another language-aware editor. Second, the architectures of existing language-aware editors are substantially different from the one we describe in this section. As our primary example, continuous parsing in Eclipse occurs by re-invoking a noncontinuous but incremental parser as edits are made. Although Eclipse's approach is more difficult to implement than the continuous compiler approach we use, they can manage parsing state at a very fine granularity, which potentially allows their approach to be more scalable.

The worst-case time performance of a language-aware editor described in this section is related to the depth of the parse tree of the code being edited. As with Wagner's incremental parsing algorithm [43], performance is seriously affected by how the grammar is specified. Performance is very bad when low-level Brakus Normal Forms (BNFs) are used to describe a language, because BNFs must describe repetition through recursion, rather than repetition operators, which leads to very deep parse trees. Worst-case space performance is related to the size of the code being edited because a constant number of event handlers are installed on every node in the parse tree.

# CHAPTER 4

# SEMANTICS, SYNTAX, AND IMPLEMENTATION

To simplify our presentation of SuperGlue's semantics and implementation, SuperGlue's abstractions can be divided into multiple language layers, which are illustrated in Figure 4.1. These layers are described as follows:

- The *core layer*, which is SuperGlue's lowest layer, supports discrete rule-based reasoning. When this layer is considered alone, SuperGlue resembles a simple deterministic declarative programming language: programs are expressed as rule-like connections, which are discretely evaluated to compute relations that describe how component ports are connected.

- The *object layer* contains SuperGlue's object-oriented abstractions, which are used to organize component ports and prioritize connections. The object layer is not heavily involved in program evaluation, and most of its semantics can be expressed by translation into the core layer.

- The *signal layer* extends the core layer to support the *continuous evaluation* of signal expressions, which ensures that program behavior is automatically consistent with current signal values. Continuous evaluation is SuperGlue's most complicated feature because it involves nontrivial semantics and complicated implementation details.

- Finally, the *stream layer* contains SuperGlue's support for streams and closures, which enable imperative programming in SuperGlue. By enabling the manipulation of discrete events, the stream layer provides a robust escape mechanism from the pure declarative programming that is supported by the core and signal layers.

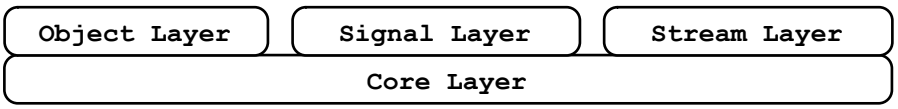


**Figure 4.1**. An illustration of how SuperGlue is layered as a language.

All of the nonroot layers are built on top of the core layer and almost always interact through the core layer, with some minor exceptions. For example, signal expressions can be transformed into event streams through the **`on`** (**`begin`**) and **`on`** (**`end`**) statements.

This chapter describes the semantics of these language layers and the implementation of these semantics in our prototype. We describe semantics only informally and, when it is useful, clarify our description with pseudocode of an ideal and inefficient implementation that glosses over many details that are present in a real implementation. This pseudocode is written in a simple functional language without objects or implicit state. Types in our pseudocode language are indicated by naming convention, e.g., `e` is an expression while `c` is a connection. To describe behavior that varies according to different kinds of data types, our pseudocode language supports guarded dispatch on argument types. For example, if `e.p` is a port expression and $\texttt{e}_0$ `+` $\texttt{e}_1$ is an arithmetic expression, then separate definitions of an `EVAL(e)` function can resolve calls over `e.p` and $\texttt{e}_0$ `+` $\texttt{e}_1$ arguments. We also use this approach to describe structural induction. For example, if `[x|`$\overrightarrow{\texttt{xs}}$`]` is a list with at least one element and $\epsilon$ is the empty list, then separate `TRAVERSE` function definitions can resolve calls over `[x|`$\overrightarrow{\texttt{xs}}$`]` and $\epsilon$ arguments. Finally, in our pseudocode language, stateful behavior is expressed through continuation passing and iteration is expressed through recursion. Continuations and recursion are very useful when we discuss continuous evaluation in Section 4.3, because this evaluation is spread out over time and requires treating current evaluation state and position as data.

We demonstrate how SuperGlue's semantics are implemented with a prototype implementation that we have constructed. Our prototype of SuperGlue is an unsophisticated interpreter that is implemented in around 4,000 lines of Java code and 500 lines of

ANTLR (parser) code. Our prototype implements SuperGlue's semantics in the most direct way possible, which significantly hurts its performance. For example, we have found that for primitive operations, SuperGlue is on the order of 100 times slower than equivalent Java code. This performance is adequate for programs that execute glue code only very rarely, which include many user-interface programs. As a result, our prototype's performance is adequate to execute the real programs in Chapter 3.

The rest of this chapter is organized as follows. The semantics and implementation of SuperGlue's core, object, signal, and stream language layers are presented in Section 4.1, Section 4.2, Section 4.3, and Section 4.4, respectively. The last two sections of this chapter describe issues related to the entire language. Section 4.5 describes SuperGlue's performance issues. Section 4.6 presents the syntax for the SuperGlue language and describes SuperGlue's syntactic sugar.

## 4.1 Core Layer

SuperGlue's core layer defines core connection and circuit abstractions, which are used to create relations between component ports. The core layer also supports the *discrete evaluation* of connections and circuits, which computes the value of a port at a single point in time. During a single time point, all signal values are frozen, and mutations do not need to be considered. Changes in mutable state are handled by the signal layer, which is presented in Section 4.3. The rest of this section details connections, circuits, and discrete evaluation.

### 4.1.1 Connections

A connection in SuperGlue is a rule with a consequent that expresses an equivalence relationship between two ports (signal or stream) and antecedents that are conditions that guard when the connection can be used. A connection is expressed in the context of a class implementation, which we say "contains" the connection. The port relationship established by a connection (left-hand side) must be to a *sink*, which is one of the following kinds of ports:

– An import of an object created in the containing class implementation;

- An import accessed through a connection variable whose type is a class;
- An import accessed through a connection variable whose type is the inner object of an object created in the containing class implementation;
- An import accessed through a connection variable whose type is the inner object accessed through another connection variable, where the type of this latter connection variable is a class;
- An export of the object being implemented in the containing class implementation, which is accessed through **`this`**; and
- An export accessed through a connection variable whose type is an inner object of the object being implemented (**`this`**).

All nonsink ports are *sources*. Source signals can be used in expressions, which can be used as conditions or connected to sink signals. Source streams can only be used in the **`on`**, **`do`**, or **`for`** clauses, and can only be connected to similar kinds of sink streams, e.g., a source command stream can only be connected to a sink command stream.

A connection can be guarded by multiple conditions that are either boolean expressions or connection queries. We describe connection queries in Section 4.2.3. A connection is guarded by all conditions expressed in any **`if`** clauses that enclose the connection's consequent up until a closure boundary, which is indicated by a stream access (Section 4.4.1). The conditions of multiple nested **`if`** clauses are conceptually conjoined together in order. For example, "**`if`** `(a)` **`if`** `(b)` **`foo.`**`x = y;`" is equivalent to "**`if`** `(a && b)` **`foo.`**`x = y;`." Besides conjunction, conditions that are boolean expressions can be composed through negation and disjunction.

Connections are type checked according to the primitive and class types of connection's sink and source. Inner object and interface types are never examined by the static type checker, because their incompatibilities are checked and resolved at run-time. Primitive types must match exactly during static type checking. For example, only an integer expression can be connected to an integer sink signal. For class types, the source type of a connection can be a class that extends the sink type of the connection. For

example, an expression of class type `Foo` can be connected to a sink signal of class type `Bar` if class `Foo` extends class `Bar`. An argument binding is type checked in the same way, where the type of the argument expression is the source type and the argument's expected type is the sink type of the connection.

In our pseudocode implementation, the structure of a signal connection is as follows:

```
if (⃗d) e
```

In this pseudocode, `e` identifies an expression, `d` identifies a condition (boolean expression or connection query), and the right arrow over bar ($\overrightarrow{\cdots}$) expresses a list, i.e., $\overrightarrow{\mathtt{d}}$ is a list of conditions. As a simplification, we do not consider signals that require arguments, which require more verbose semantics but do not fundamentally affect our discussion. The sink of a connection is not expressed inside the connection itself. Instead, the sink of multiple connections are specified in circuits as described in Section 4.1.2. The source expression of the connection is `e`.

A connection is *active* if and only if all of its conditions evaluate to true. Discrete evaluation can determine if a connection is active at some point in time. As a connection's conditions are evaluated, variables can be bound through connection queries. These variable bindings are available to successive conditions and can ultimately be used to evaluate the connection's source expression. A variable binding environment is expressed as $\Theta$. In our pseudocode, an active connection is described by the `ACTIVE(Θ, c)` function, which is defined over a variable binding environment ($\Theta$) and a connection (`c`) as follows:

```
ACTIVE(Θ, if ([d|⃗dX]) e) =
  let <Θ', v> = NOW_COND(Θ, d) in
    if (v == false) <Θ, false>;
    else ACTIVE(Θ', if ⃗dX e)

ACTIVE(Θ, if (ϵ) e) = <Θ, true>;
```

In our pseudocode language, the `[d|⃗dX]` cons operator matches nonempty lists, where the first element is bound to `d` and a sublist that does not include the first element is bound to $\overrightarrow{\mathtt{dX}}$; and $\epsilon$ is the empty list. This is similar to how unification occurs in Prolog [36]. The `NOW_COND(Θ, d)` function, which will be defined later, evaluates a

condition and therefore returns two values: a new variable environment (Θ') that can convey new variable bindings, and the boolean result of the evaluation (`v`). Pair values are expressed inside angle brackets; e.g., `<Θ, false>` indicates a pair with Θ as its first value and `false` as its second value. A **`let`** clause can be used to bind single or pair values to variables, e.g., **`let`** `<Θ', v> = ...` binds the values of a pair to Θ' and `v`. The `ACTIVE` function returns a variable binding environment (Θ), which indicates how variables are bound if the connection is active, and a truth value that indicates if the connection is active. The evaluation of conditions that are connection queries is described in Section 4.2.3. Conditions that are expressions are evaluated according to the following pseudocode:

```
NOW_COND(Θ, e) = <Θ, NOW_EXPR(Θ, e)>;
```

The $\texttt{NOW}_{\texttt{EXPR}}(\Theta, \texttt{e})$ function evaluates an expression. Because variables never become bound in expressions, the $\texttt{NOW}_{\texttt{EXPR}}$ function only returns the current value of the evaluated expression.

The discrete evaluation of primitive expressions in SuperGlue occurs in the traditional way. For example, the pseudocode $\texttt{NOW}_{\texttt{EXPR}}(\Theta, \texttt{2 + 2})$ evaluates to `4`. The discrete evaluation of an expression of the form `e.p`, where `p` is a signal that is implemented as a circuit, involves locating and evaluating circuits according to semantics that are described in Section 4.1.2. The discrete evaluation of binary boolean operation short circuits, meaning the second operand of the operation is not evaluated if the operations value can be determined by the value of the right operand. Short circuiting is also mandatory for the evaluation of a connection's conditions because they are conjoined.

For a signal implemented as a Java driver, discrete evaluation calls the `now()` driver method (described in Section 2.4.2) to access its value. We express this evaluation in pseudocode as follows:

```
NOW_EXPR(Θ, e.j) = NOW_JAVA(NOW_EXPR(Θ, e), j);
```

In this pseudocode, `j` indicates a signal that is implemented with a Java driver. Because the $\texttt{NOW}_{\texttt{JAVA}}$ function calls a Java method, it is a base case that cannot be defined in pseudocode.

### 4.1.2 Circuits

A circuit in SuperGlue is an organization of connections that connect to the same port in an object or inner object. Connections are prioritized in a circuit according to the types of their targets or through explicit **else** clauses. Connections that appear under the scope of an **if** clause have priority over connections that appear under the scope of a corresponding **else** clause. Unfortunately, our current interpretation of else-based prioritization is not similar to **else** clauses in languages with explicit control flow: a connection that appears under an *else* clause is not necessarily guarded by the negation of the conditions in the corresponding **if** clause. Such nonintuitive behavior occurs when **if** and **else** clauses are nested. As an example, consider the following code:

```
if (clock.time % 2 == 0) {
  if      (model.temperature > 90)
    view.color = red;
  else if (model.temperature < 40)
    view.color = blue;
} else view.color = black;
```

When time is even, the view is black if the current temperature is between `40` and `90`. This behavior is counter-intuitive to standard if-else semantics, where the view's color would be unconnected. We chose priority-only else clause semantics because it can be implemented more efficiently than standard if-else semantics and behaves as expected in common cases. However, because standard if-else semantics are always more intuitive, we should change this in the future when better compilation techniques can be used to mitigate the performance penalty of standard if-else semantics.

The structure of a port's circuit is a two-dimensional list of connections, or $\overrightarrow{\overrightarrow{\texttt{c}}}$, where `c` is a connection of the structure described in Section 4.1.1. The first dimension of this list is ordered according to connection priority, where lists of connections with higher priorities appear earlier in this list. The second dimension of this list is unordered and contains connections with the same priority. For example, the circuit `[[`$\texttt{c}_1$`, `$\texttt{c}_2$`], [`$\texttt{c}_3$`, `$\texttt{c}_4$`, `$\texttt{c}_5$`], [`$\texttt{c}_6$`]]`, which is illustrated in Figure 4.2, contains six connections. In this circuit, the $\texttt{c}_1$ and $\texttt{c}_2$ connections have the same priority, which is the highest in

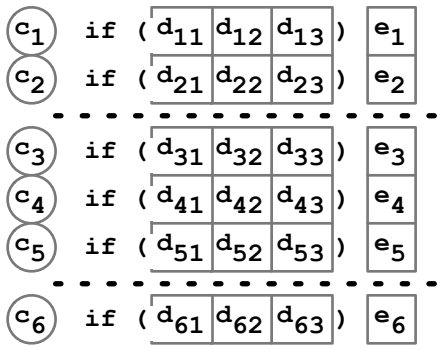


**Figure 4.2**. An illustration of a circuit: connections are labeled in circles, dashed lines indicate priority boundaries.

the circuit. The $c_3$, $c_4$, and $c_5$ connections also have the same priority, which is in the middle of the circuit. Finally, the $c_6$ connection has the lowest priority in the circuit.

Each object and inner object contains a circuit for each of their ports that is not implemented as a driver. Evaluating the expression `e.p` involves evaluating `e` into an object or inner object value (`o`) and locating the circuit for `p` in this value, which is expressed by the pseudocode for the $\texttt{NOW}_{\texttt{EXPR}}(\Theta,\ \texttt{e.p})$ function in Figure 4.3. In this $\texttt{NOW}_{\texttt{EXPR}}$ function, the `FIND-CIRCUIT(v, p)` function (pseudocode not shown) is used to locate a circuit given an object or inner object value and a port. The $\texttt{NOW}_{\texttt{CIRC}}(\overrightarrow{\texttt{c}})$ function (pseudocode shown) is then used to find the current value of the circuit.

The behavior of the $\texttt{NOW}_{\texttt{CIRC}}$ function, which is defined by pseudocode in Figure 4.3, is best illustrated through an example. Figure 4.4 illustrates how the circuit in Figure 4.2 can undergo discrete evaluation. The $\texttt{NOW}_{\texttt{CIRC}}$ function traverses each connection until it locates an active connection, which is determined by the `ACTIVE` function that was defined in Section 4.1.1. In Figure 4.4, the circuit's $c_1$ connection is checked first and is found not to be active because its $d_{12}$ condition evaluates to false. Likewise, the circuit's next $c_2$ and $c_3$ connections are found not to be active because they have conditions that evaluate to false. The $c_4$ connection is found to be active because all its conditions evaluate to true.

The $\texttt{NOW}_{\texttt{CIRC}}$ function defined in Figure 4.3 can get "stuck," meaning it can be undefined. First, the $\texttt{NOW}_{\texttt{CIRC}}$ function gets stuck on circuits with no remaining con-

```
NOW_EXPR(Θ, e.p) =
  let <Θ', if (d⃗) e'> =
        NOW_CIRC(FIND-CIRCUIT(NOW_EXPR(Θ, e), p))
  in NOW_EXPR(Θ', e');

NOW_CIRC([c|[cX⃗|cY⃗⃗]]) =
 let <Θ, v> = ACTIVE(ϵ, c) in
    if (!v) NOW_CIRC([cX⃗|cY⃗⃗]);
    else if (NOT-ACTIVE(cX⃗)) <Θ, c>;

NOW_CIRC([ϵ|cY⃗⃗]) = NOW_CIRC(cY⃗⃗);

NOT-ACTIVE(Θ, [c|cX⃗]) =
  let <Θ', v> = ACTIVE(Θ, c) in !v && NOT-ACTIVE(Θ, cX⃗);

NOT-ACTIVE(Θ, ϵ) = true;
```

**Figure 4.3**. The discrete evaluation of a signal expression in SuperGlue.

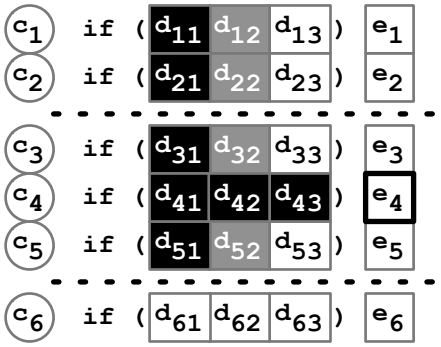


**Figure 4.4**. An illustration of discrete evaluation over the circuit that was illustrated in Figure 4.2: white-on-black conditions have evaluated to true, white-on-gray conditions have evaluated to false, black-on-white conditions have not been evaluated, and a bold black box is used to indicate the source expression whose evaluation is used as the circuit's current value.

nections to traverse, which means the targeted port is unconnected. Second, the $\texttt{NOW}_{\texttt{CIRC}}$ function gets stuck if more than one connection of the same priority is active, which means the targeted port is ambiguously connected. For example, in Figure 4.4, even though the $\texttt{NOW}_{\texttt{CIRC}}$ function has found the $\texttt{C}_4$ connection to be active, it must still traverse connection $\texttt{C}_5$ to check if the targeted signal is connected unambiguously. In the pseudocode definition of the $\texttt{NOW}_{\texttt{CIRC}}$ function in Figure 4.3, ambiguity is checked for with the $\texttt{NOT-ACTIVE}$ function, which returns true if no other connection that shares the active connection's priority is active. Connection errors are easy to detect at run-time, but are very difficult to detect statically. As a result, our semantics do not require the static detection of connection errors, and our prototype only detects connection errors at run-time. Recovery in the presence of a connection error (unconnected or ambiguous) is not currently supported by SuperGlue's semantics. When a connection error is detected, our prototype simply terminates the program. SuperGlue currently lacks the abstractions to adequately recover from connection errors.

As with the evaluation of a connection's conditions, the evaluation of a circuit's connections also short circuits: a lower priority connection is not evaluated unless every higher priority connection is not active. For example, the lowest priority $\texttt{C}_6$ connection in Figure 4.4 is not evaluated at all because the active $\texttt{C}_4$ connection has a higher priority. As a result, higher priority active connections can guard against the erroneous evaluation of lower priority connections. On the other hand, if a connection of one priority is evaluated, then all other connections in the circuit of the same priority are also evaluated. This behavior is necessary to detect ambiguity.

### 4.1.3 Implementation

Our current prototype implementation of SuperGlue's discrete evaluation traverses circuits and connections in a way that is similar to the functions in this section. Given its simple semantics, discrete evaluation is also amenable to compilation, which we describe in Section 6.1.

## 4.2 Object Layer

SuperGlue's object layer is primarily concerned with the organization of connections according to extensible object types. The object layer enhances SuperGlue with classes, inner objects, and interfaces. Classes are object templates that enable object implementation reuse through class extension; inner objects allow objects to contain an unbounded number of ports; and interfaces enable the reuse of compound port types. Class, inner object, and interface types can be used to declare connection variables, which allow programs to abstract over connections. Values can be connected to inner objects, which allows port connections to be analyzed at run-time through connection queries. The combination of inner objects, connection variables, and connection queries allow connection graphs to grow in complex and unbounded ways according to only a small number of rule-like connections.

The object layer has little effect on the evaluation semantics defined in the core, signal, and stream layers. Most of the object layer's semantics can be described as a translation from object layer code into core layer code. The rest of this section describes the semantics and implementation details of abstractions in the object layer.

### 4.2.1 Inner Object

A inner object is created on demand by an object implementation to import or export additional ports. Creation of an inner object is straightforward: SuperGlue or Java driver code specifies a value that is connected to a new inner object. An inner object has no identity: its imports can only be connected through connection variables, whose semantics are described in Section 4.2.2, and an inner object can only be distinguished from other inner objects of similar types through connection queries, whose semantics are described in Section 4.2.3. As a result, the semantics of an inner object are closely related to these two abstractions.

### 4.2.2 Connection Variables

Connection variables parameterize connections in a way that is similar to how arguments parameterize procedures. The type of a connection variable determines what

objects and inner objects a referring connection applies to. Given an object or inner object value of type $S$ and a connection that targets a port in a connection variable of type $T$, then this connection applies to the value if $S$ is a subtype of $T$. Conceptually, each object or inner object value is associated with its own set of circuits, where target connection variables are replaced with the object or inner object value. For example, if `o` is an object of type $T$, `v` is a connection variable of type $S$, and `v.p = x` is a connection, then the connection `o.p = x` belongs to the circuit for port `p` in object `o`.

Because connection variables can be eliminated in a connection through replacement, evaluation semantics do not need to consider them. However, the type of a connection variable is used to prioritize a referring connection in a circuit in a way that is described Section 4.2.4. To conserve memory in our prototype, connections are not duplicated for every object and inner object value that is created. Instead, connection variable replacement occurs as parameter binding when a connection is evaluated.

### 4.2.3 Connection Queries

Connection queries are used to query at run-time what value is connected to an inner object. Because connection queries bind variables, they can only be used as expressions in conjoined conditions in connections, i.e., connection queries cannot be negated or used in disjunctions. The discrete evaluation semantics of a connection query is defined in the $\texttt{NOW}_{\texttt{COND}}$ function as follows:

```
NOW_COND(Θ, u = <T> e) =
  QUERY(Θ, NOW_EXPR(Θ, e), u, T);

QUERY(Θ, v, u, T) =
  if (ISA(v, T)) <BIND(Θ, u, v), true>;
  else if (!HAS-FROM(v)) <Θ, false>;
  else QUERY(Θ, GET-FROM(v), u, T);
```

The `QUERY` function performs the connection query after the queried expression has been discretely evaluated. The `HAS-FROM` function returns true only if it is called over an inner object value that is connected from another value; and the `GET-FROM` function returns this value. The `BIND` function (pseudocode not shown) creates a new variable

binding environment that combines a specified variable binding with an existing variable binding environment.

Beyond having evaluation semantics, connection queries are also involved in the prioritization of their containing connections. Prioritization through connection queries is very useful because often inner objects of the same type must behave differently depending on what values are connected to them. We discuss prioritization with connection queries in Section 4.2.4.

Finally, equality statements in SuperGlue are connection queries that determine if a value is or is connected to another value. With these semantics, inner objects are truly identified by the values that they are connected from.

### 4.2.4 Type-based Prioritization

Besides the else-based prioritization described in Section 4.1.2, connections can also be prioritized by the declaration types of the connection variables that contain their target ports. Type-based prioritization in SuperGlue is based on subtyping: if the target type of connection $c_a$ is $S$ and the target type of connection $c_b$ is $T$, and $S$ is a subtype of $T$, then $c_a$ has priority over $c_b$ in a circuit. These semantics are similar to how method overriding occurs in other object-oriented languages. The main difference here is that the declared target types involved can interface or inner object types in addition to class types.

The subtyping rules used to prioritize connections in circuits are shown in Figure 4.5. Most of these rules express conventional subtyping behavior based on transitivity, class, interface, and inner object extension relationships (`extends`), and interface import and export relationships (`imports` and `exports`). Priority subtyping for inner objects is tricky because it must account for extension of both the inner object type and the containing class. Connections that are expressed over more specific class types must have priority over connections expressed over more specific inner object types because this is the only way to override inner object behavior. Therefore, the rules in Figure 4.5 ensure that class extension is considered before inner object extension. Finally, the last rule in Figure 4.5 states that an object is a type that is a subtype of its creating class. In

$$\frac{\Delta \vdash_t T_x < T_y, T_y < T_z}{\Delta \vdash_t T_x < T_z} \qquad \frac{T \texttt{ extends } S \in \Delta}{\Delta \vdash_t T < S}$$

$$\frac{T \texttt{ imports } I \in \Delta}{\Delta \vdash_t T < I} \qquad \frac{T \texttt{ exports } I \in \Delta}{\Delta \vdash_t T < I}$$

$$\frac{\Delta \vdash_t C < D, F <= E \quad F, E \in \texttt{inner}(D, \Delta)}{\Delta \vdash_t C.F < D.E} \qquad \frac{F \texttt{ extends } E \in \Delta \quad F \in \texttt{inner}(C, \Delta)}{\Delta \vdash_t C.F < C.E}$$

$$\frac{o = \texttt{new } C \in \Delta}{\Delta \vdash_t o < C}$$

**Figure 4.5**. Rules that define priority subtyping between class, interface, and inner object types; $\Delta$ is the typing environment.

other words, connections expressed directly to an object have priority over connections that are expressed through connection variables.

When the target types of two connections are the same, then the types of connection queries over their target connection variables can be used to prioritize the connections. The same subtyping rules in Figure 4.5 are used to determine how the connection query types prioritize the connections. If one of these connections does not contain a connection query, then an implicit "any" type is assumed so that the other connection that is guarded by a connection query has priority over it. If a connection has multiple connection queries over its target connection variable, then only if one connection query type in one connection is a subtype of another connection query type in the other connection does the former connection have priority over the latter connection.

Type-based prioritization works across different class implementations when connecting exports. As a result, the implementation of a subclass can override how the export of a superclass is connected. SuperGlue currently lacks the three mechanisms needed to effectively manage this behavior: a "final" mechanism that can prevent subclasses from overriding how an export is connected, a "super" mechanism that can be used to

access a lower priority export connection specified in a superclass, and a "protected" mechanism to specify exports that can only be accessed in subclasses. Figuring out how these mechanisms work in SuperGlue should be straightforward, and they should be added to a future revision of the language.

Because else-based prioritization always occurs locally within one source file, it is considered before type-based prioritization, which can occur globally across multiple source files and modules. In other words, **`else`** statements can always be used to override object-oriented prioritization for connections that are expressed in the same source file.

### 4.2.5 Interfaces

As described in Figure 4.5, interface types are involved in the type-based prioritization of connections in circuits. Because a class or inner object can import or export multiple interfaces, it is possible for two connections with different target types that include the same target to have no type-based priority relationship with each other. In this case, the connections exist in the same priority bank of a circuit unless else-based prioritization is used.

Interfaces do not specify if their ports are imported or exported. Instead, whether an interface's ports are imported or exported is specified by the implementing class or inner object type. This now leads to the following issue of when interfaces are specified as types in connection variables and connection queries: should the resulting variables import or export the interface's ports? Because exports are connected through the **`this`** keyword, connection variables are always used to connect imports. As a result, a connection variable with an interface type is bound to values that import the ports of this interface. On the other hand, imports can never be connected through a variable defined by a connection query. As a result, a connection query variable with an interface type is bound to values that export the ports of this interface.

Special connections are automatically added to the circuit of an inner object that implements the interface and contains the interface's ports as sinks. These connections handle the case where a value that is connected to the inner object contains the interface's

ports as source ports. In this case, the source interface ports of the value can be connected to the sink interface ports of the inner object. All other connections have a higher priority than these connections.

### 4.2.6 Modules

Connections can be packaged into a module so that they can be reused in multiple programs. When a program depends on a module, connections specified in that module are automatically added to the program. This behavior allows a module to provide default connections to the imports of the classes it defines. For example, the GlueUI module defines the `TableView` class and specifies the default connection of connecting the empty list to the `rows` import of `TableView` objects. Modules can be used to reuse adapter connections that integrate two unrelated modules. For example, a `mail.glueui` module can contain connections that specify how email objects are manipulated in a user interface.

A module's dependencies with other modules are explicitly specified in the module's definition, and a module can only refer to definitions from these modules. As with Java's package system, a module cannot automatically refer to definitions contained in any indirect dependencies. Likewise, a class implementation contains connections that are specified in its implementation, the module that contains it, and the modules that are directly used by this module. A class implementation does not contain connections in modules that it indirectly depends on. For example, if module `a` directly depends on module `b`, and module `c` directly depends on module `b`, then a class implementation in module `c` can use definitions from module `b` but not module `a`. Additionally, any connection specified in module `b`, but not module `a`, is included in the circuits of the class implementation. This design decision is somewhat controversial because one might expect that indirect module dependencies would be treated the same as direct module dependencies. However, because we are emulating Java's package system with our design, indirect module dependencies are ignored.

## 4.3 Signal Layer

Continuous evaluation ensures that components are always consistent with the values of the state that they are viewing. True continuous evaluation cannot be implemented efficiently with discrete computations. Instead, continuous evaluation is often simulated with event handling, where components are notified through events of changes in state after the changes have occurred. When compared to discrete evaluation, SuperGlue's implementation of continuous evaluation is substantially more complicated because it must manage event handling behavior. This complexity represents a significant amount of programming effort that would be replaced by repetitive code if performed manually.

Because SuperGlue uses event handling to propagate changes in state, its support for continuous evaluation is asynchronous, where components' views of state can be inconsistent while state changes are propagating through the system. Asynchronous change propagation is not very clean semantically and leads to atomicity issues that must be addressed in our implementation and Java-based component drivers. We describe these atomicity issues in Section 4.3.2. Some functional-reactive programming languages, such as Yampa [20], support synchronous continuous evaluation, which is less efficient but semantically cleaner (no atomicity issues) than asynchronous continuous evaluation. We discuss Yampa as related work in Section 5.1.1.

A Java-based component driver initiates continuous evaluation on an imported signal's circuit by installing a Java-based *target* event handler on the circuit. The target event handler of a signal access is called when the signal's value changes. The installation of a target event handler involves installing *switching* event handlers on the conditions of the circuit's connections that detect when the circuit's connections become active and inactive. Switching event handlers perform the switching behavior that changes what source expression a target event handler is installed on. As an example, the circuit from Figure 4.2 undergoes continuous evaluation in Figure 4.6. To support short-circuiting semantics, switching event handlers are installed on conditions that do not follow false conditions in connections that are not lower-priority than an active connection. For example, switching event handlers are installed on the $\mathtt{d}_{11}$ and $\mathtt{d}_{12}$ conditions in the $\mathtt{c}_1$ circuit, but because the $\mathtt{d}_{12}$ condition is false, a switching event handler is not installed

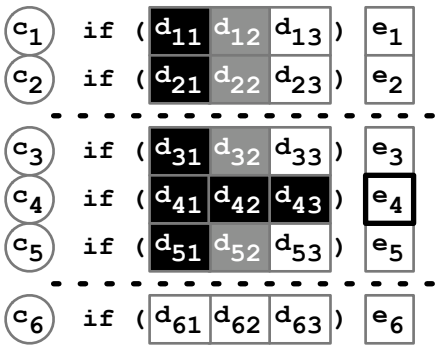


**Figure 4.6**. An illustration of the circuit in Figure 4.2 undergoing continuous evaluation: switching handlers are installed on white-on-black conditions (which currently evaluate to true), switching handlers are installed on white-on-gray conditions, which currently evaluate to false, black-on-white conditions have not been evaluated and switching handlers are not installed on them, the target event handler is installed on the connection source expression that is surrounded by a bold box.

on the $\mathtt{d}_{13}$ condition. Also, switching conditions are not installed on any conditions in the $\mathtt{c}_6$ connection because the $\mathtt{c}_4$ connection is active.

As with discrete evaluation, continuous evaluation fails if a signal becomes unconnected or connected ambiguously. To detect when state changes cause a circuit to become connected ambiguously, switching event handlers are installed on conditions in connections that have the same priority as the found active connection. In Figure 4.6, switching event handlers are installed on conditions in the $\mathtt{c}_3$ and $\mathtt{c}_5$ connections because they have the same priority as the active $\mathtt{c}_4$ connection. Unconnected and ambiguous connections detected either during discrete or continuous evaluation are treated as programming errors and cause our prototype to terminate execution immediately.

Switching occurs during continuous evaluation when one of the conditions of the highest-priority active connection becomes false. In this case, this connection *yields* to an active connection of a lower priority, meaning that the target event handler is uninstalled on the higher-priority connection's source expression and installed on the lower-priority connection's source expression. As an example, the continuous evaluation of the circuit in Figure 4.6 evolves to Figure 4.7 when the $\mathtt{d}_{42}$ condition in the $\mathtt{c}_4$ connection becomes false, which is detected by its installed switching event handler.

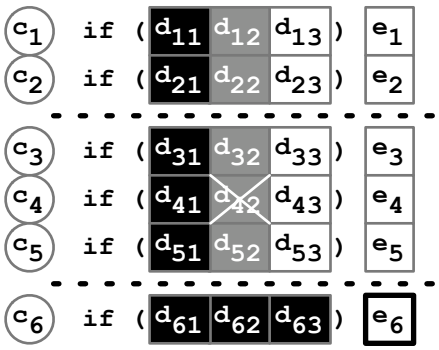


**Figure 4.7**. An illustration of the continuous evaluation in Figure 4.6 after the $\mathtt{d}_{42}$ condition in the $\mathtt{c}_4$ connection becomes false: an 'x' over a condition indicates that the condition has just become false.

Two interesting behaviors occur in this figure. First, the switching event handler that was installed on the $\mathtt{d}_{43}$ condition is uninstalled because $\mathtt{d}_{43}$ cannot be evaluated if $\mathtt{d}_{42}$ is false. Second, switching event handlers are installed on the conditions of the $\mathtt{c}_6$ connection, which is also found to be the new highest-priority connection of this circuit. As a result, the target event handler is uninstalled on the $\mathtt{e}_4$ source expression and installed on the $\mathtt{e}_6$ source expression.

Switching occurs during continuous evaluation when all of the conditions of a connection that has a higher-priority than the current highest-priority active connection become true. In this case, the higher-priority connection *preempts* the lower-priority connection to become the circuit's new highest-priority active connection. As an example, the continuous evaluation of the circuit in Figure 4.6 evolves to Figure 4.8 when the $\mathtt{d}_{12}$ condition in the $\mathtt{c}_1$ connection becomes true, which is detected by its installed switching event handler, and the $\mathtt{d}_{13}$ condition is already true when $\mathtt{d}_{12}$ is true. Three interesting behaviors occur in this figure:

– The switching and ambiguity event handlers installed on the conditions of the $\mathtt{c}_3$, $\mathtt{c}_4$, and $\mathtt{c}_5$ connections are uninstalled because these connections follow an active connection of a higher priority;

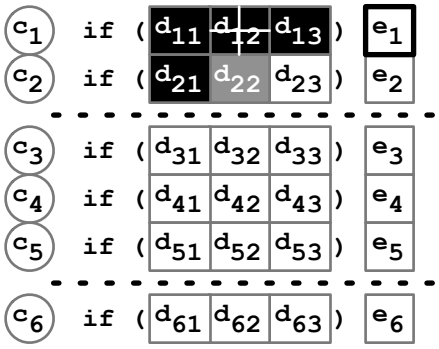


**Figure 4.8**. An illustration of the continuous evaluation in Figure 4.6 after the $\mathtt{d}_{12}$ condition in the $\mathtt{c}_1$ connection becomes true; a '+' over a condition indicates that the condition has just become true.

– A switching event handler is installed on the $\mathtt{d}_{13}$ condition because the $\mathtt{d}_{12}$ condition is now true and $\mathtt{d}_{13}$ can now be evaluated; and

– The target event handler is uninstalled on the $\mathtt{e}_4$ source expression and installed on the $\mathtt{e}_1$ source expression.

Because conditions in SuperGlue can refer to signal expressions, the installation of a switching event handler on a connection condition can initiate continuous evaluation on other circuits. In this recursive case, the switching event handler becomes the target event handler of this continuous evaluation. As a base case, the continuous evaluation of a signal that is implemented as a Java-based driver occurs by calling the driver's `install()` and `uninstall()` methods. In the same way, recursive continuous evaluation also occurs when target event handlers are installed on connection source expressions.

A Java-based component driver terminates continuous evaluation on an imported signal's circuit by uninstalling the event handler it originally installed on an imported signal's circuit. Termination of continuous evaluation involves uninstalling all switching and ambiguity event handlers on connection conditions and uninstalling the target event handler on the source expression of the current highest-priority active connection. Unin-

stallation of an event handler on expression can recursively terminate other continuous evaluations as well.

### 4.3.1 Pseudocode

The pseudocode in Figure 4.9 defines the $\texttt{INSTALL}_{\texttt{CIRC}}(\Sigma, \Theta, \overrightarrow{\texttt{c}}, \texttt{h})$ function, which initiates continuous evaluation on a circuit ($\overrightarrow{\texttt{c}}$) by installing a target event handler (h) on that circuit. The $\texttt{INSTALL}_{\texttt{CIRC}}$ function finds the highest priority connection of a circuit by traversing each of the circuit's connections, and each of a connection's conditions. As in discrete evaluation, traversal of a connection's conditions stops when a false condition is encountered, and traversal of a circuit's connections stops when an active connection is detected. Unlike discrete evaluation, the $\texttt{INSTALL}_{\texttt{CIRC}}$ function installs switching event handlers on connection conditions that are instances of the $\textit{handle}_{\textit{switch}}$ constructor. Switching event handlers track the current binding environment ($\Theta$), the current position of the evaluation ($[[\textit{if}\ ([\texttt{d}|\overrightarrow{\texttt{dX}}])\ \texttt{e}|\overrightarrow{\texttt{cX}}]|\overrightarrow{\overrightarrow{\texttt{cY}}}]$), and the target event handler (h) that is being installed. When switching event handlers are dispatched, they use these values to recreate continuous evaluation context. The pseudocode in Figure 4.9 also defines the $\texttt{INSTALL}_{\texttt{AMBG}}(\Sigma, \Theta, \overrightarrow{\texttt{c}}, \texttt{h})$ function, which installs ambiguity event handlers on connection conditions that follow the highest-priority active connection. Ambiguity event handlers are instances of the $\textit{handle}_{\textit{ambg}}$ constructor.

Besides a variable binding environment, the $\texttt{INSTALL}_{\texttt{CIRC}}$ and $\texttt{INSTALL}_{\texttt{AMBG}}$ functions also manipulate an event handling environment ($\Sigma$), which tracks what event handlers are installed on Java-based drivers. In our pseudocode, the event handling environment is updated as a continuation: INSTALL functions create new event handling environments with additional installed event handlers, while the UNINSTALL functions create new event handling environments where existing event handlers are no longer installed.

The $\texttt{INSTALL}_{\texttt{EXPR}}(\Sigma, \Theta, \texttt{e}, \texttt{h})$ function used in Figure 4.9 installs an event handler (h) that listens for changes in an expression (e). The $\texttt{INSTALL}_{\texttt{EXPR}}$ function is defined over expressions to listen for changes in operand expressions and to recompute

```
INSTALL_CIRC(Σ, Θ, [[if ([d|dX⃗]) e|cX⃗]|cY⃗⃗], h) =
  let h′ = handle_switch(Θ, [[if ([d|dX⃗]) e|cX⃗]|cY⃗⃗], h);
  in let Σ′ = INSTALL_EXPR(Σ, Θ, d, h′);
  in let <Θ′, v> = NOW_COND(Θ, d);
  in if (v == false) INSTALL_CIRC(Σ′, ϵ, [cX⃗|cY⃗⃗], h);
  else INSTALL_CIRC(Σ′, Θ′, [[if (dX⃗) e|cX⃗]|cY⃗⃗], h);

INSTALL_CIRC(Σ, Θ, [[if (ϵ) e|cX⃗]|cY⃗⃗], h) =
  INSTALL_EXPR(INSTALL_AMBG(Σ, ϵ, cX⃗), Θ, e, h);

INSTALL_CIRC(Σ, Θ, [ϵ|cY⃗⃗], h) = INSTALL_CIRC(Σ, ϵ, cY⃗⃗, h);

INSTALL_AMBG(Σ, Θ, [if ([d|dX⃗]) e|cX⃗]) =
  let h = handle_ambg(Θ, [if ([d|dX⃗]) e|cX⃗])
  in let Σ′ = INSTALL_EXPR(Σ, Θ, d, h)
  in let <Θ′, v> = NOW_COND(Θ, d)
  in if (v == false) INSTALL_AMBG(Σ′, ϵ, cX⃗);
  else INSTALL_AMBG(Σ′, Θ′, [if (dX⃗) e|cX⃗]);

INSTALL_AMBG(Σ, Θ, [if (ϵ) e|cX⃗]) = INSTALL_AMBG(Σ, ϵ, cX⃗);

INSTALL_AMBG(Σ, Θ, ϵ) = Σ;
```

**Figure 4.9**. Pseudocode for the $\texttt{INSTALL}_{\texttt{CIRC}}$ and $\texttt{INSTALL}_{\texttt{AMBG}}$ event handler installation functions, which are respectively used to detect changes that cause a circuit to become active or to be connected ambiguously: $\Sigma$ is the event handling environment, which internally maintains what event handlers are installed on signals implemented as drivers.

the value of the expression based on the operation's semantics. For example, the pseudocode $\texttt{INSTALL}_{\texttt{EXPR}}(\Sigma,\ \Theta,\ \texttt{e + 2},\ \texttt{h})$ evaluates to an event handling environment where `h` is notified of a new value for `e + 2` whenever the value of `e` changes. Installing an event handler on an expression whose value never changes simply resolves to a no-op operation where the event handling environment is unmodified. For example, the pseudocode $\texttt{INSTALL}_{\texttt{EXPR}}(\Sigma,\ \Theta,\ \texttt{2 + 2},\ \texttt{h})$ evaluates to $\Sigma$.

The $\texttt{INSTALL}_{\texttt{EXPR}}$ function is defined for signal accesses (`e.p`) with the following pseudocode:

```
INSTALL_EXPR(Σ, Θ, e.p, h) =
   INSTALL_CIRC(INSTALL_EXPR(Σ, Θ, e, handle_expr(p, h)),
```

```
FIND-CIRCUIT(NOW_EXPR(Θ, e), p), h);
```

In this pseudocode, the *handle*$_{expr}$ event handler, whose behavior will be defined later, detects when the target expression changes and re-installs the targeted event handler (h). The INSTALL$_{EXPR}$ function installs the target handler for a known object value, which involves calling the INSTALL$_{CIRC}$ function on the circuit for the target port (p) in this value.

If a signal is not defined as a circuit, then it is defined as a Java driver. Installing an event handler on a driver directly produces a new environment, which is expressed in the following code:

```
INSTALL_EXPR(Σ, Θ, e.j, h) =
  let Σ′ = INSTALL_EXPR(Σ, Θ, e, handle_expr(j, h))
  in INSTALL_JAVA(Σ′, NOW_EXPR(Σ, e), j, h);
```

The INSTALL$_{JAVA}$ function installs an event handler on a signal (j) that is implemented as a driver rather than as a circuit.

Uninstallation functions simply reverse the behavior of installation functions. For each function that begins with INSTALL, a corresponding function that begins with UNINSTALL exists with the similar code that calls UNINSTALL functions instead of INSTALL functions.

We define the AFTER(Σ, h, v) function to propagate new driver signal values to event handlers that are installed on these signal drivers. The pseudocode of this propagation is as follows:

```
NOTIFY_JAVA(Σ, o.j, v) = PROPAGATE(Σ, INSTALLED(o.j), v);

PROPAGATE(Σ, [h|hX⃗], v) =
  PROPAGATE(AFTER(Σ, h, v), hX⃗, v);

PROPAGATE(Σ, ϵ, v) = Σ;
```

The NOTIFY$_{JAVA}$ function detects a change in the value of a driver signal j in value o for some current event handling environment; the INSTALLED function returns a list of all event handlers that are installed on a signal driver; and the PROPAGATE function

is used to traverse these event handlers (producing new event handling environments as usual). The AFTER function is defined over specific kinds of event handlers.

Figure 4.10 presents pseudocode for the $handle_{switch}$ switching event handler and $handle_{ambg}$ ambiguity event handler. The $handle_{switch}$ switching event handler detects when yielding or preemption should occur in a circuit. When a condition being observed by a switching event handler becomes true, the connection either becomes active or is closer to becoming active. The $\text{ADVANCE}_{\text{COND}}(\Sigma, \Theta, \overrightarrow{c})$ function installs switching event handlers on conditions that follow the newly true condition until a false condition is encountered. If all remaining conditions in the connection are true, or there are no remaining conditions, then the $\text{ADVANCE}_{\text{COND}}$ function performs connection preemption. When a condition being observed by a switching event handler becomes true, the connection either becomes inactive or becomes farther away from being active. The $\text{ADVANCE}_{\text{COND}}$ function uninstalls switching event handlers on all conditions that follow the newly false condition until a false condition. If all following conditions in the connection were previously true, or there are no remaining conditions, then the $\text{ADVANCE}_{\text{COND}}$ function performs connection yielding.

The $handle_{expr}$ event handler listens for changes in the target of a signal expression. The following pseudocode expresses $handle_{expr}$ behavior over a signal that is implemented as a circuit:

```
BEFORE(Σ, handle_expr(p, h), v) =
  UNINSTALL_CIRC(Σ, FIND-CIRCUIT(v, p), h);

AFTER(Σ, handle_expr(p, h), v) =
  INSTALL_CIRC(Σ, FIND-CIRCUIT(v, p), h);
```

The BEFORE function (pseudocode not shown) is called before a change in a signal's value occurs, which allows us to uninstall the targeted handler (h) from the old circuit. The AFTER function is called after the change has occurred, which allows us to install the targeted event handler on the new circuit.

```
AFTER(Σ, handle_switch(Θ, [[if ([d|d⃗X]) e|c⃗X]|c⃗⃗Y], h), v) =
  let (Θ′, v′) = NOW_COND(Θ, d)
  in ADVANCE_COND(Σ, Θ′ [if (d⃗X) e|[c⃗X|c⃗⃗Y]], v, h);

ADVANCE_COND(Σ, Θ, [[if ([d|d⃗X]) e|c⃗X]|c⃗⃗Y], v, h)
 let (Σ′, v′) = NOW_COND(Σ, d)
 in let h′ = handle_switch([[if ([d|d⃗X]) e|c⃗X]|c⃗⃗Y], h)
 in let Σ′′ = if (v) INSTALL_EXPR(Σ′, Θ, h′)
              else UNINSTALL_EXPR(Σ′, Θ, h′)
 in if (v′)
  ADVANCE_COND(Σ′′, Θ, [if (d⃗X) e|[c⃗X|c⃗⃗Y]], v, h);
 else Σ′′;

ADVANCE_COND(Σ, Θ, [if (ϵ) e|[c⃗X|c⃗⃗Y]], v, h) =
  if (!v) let Σ′ = UNINSTALL_EXPR(Σ, Θ, e, h)
          in let Σ′′ = UNINSTALL_AMBG(Σ′ Θ, c⃗X)
          in INSTALL_CIRC(Σ′′, c⃗⃗Y, Θ, h);
  else if (NOT-ACTIVE(Σ, c⃗X))
    let Σ′ = INSTALL_EXPR(Σ, Θ, e, h)
    in let Σ′′ = INSTALL_AMBG(Σ′, Θ, c⃗X)
    in UNINSTALL_CIRC(Σ′′, Θ, c⃗⃗Y, h);

AFTER(Σ, handle_ambg(Θ, c⃗X), v) = ADVANCE_AMBG(Σ, Θ, c⃗X);

ADVANCE_AMBG(Σ, Θ, [if ([d|d⃗X]) e|c⃗X]) =
 let Σ′ = INSTALL_EXPR(Σ, d, handle_ambg([if (d⃗X) e|c⃗X]))
 in let (Θ′, v) = NOW_COND(Σ′ , Θ, d)
 in if (!v) Σ′;
 else ADVANCE_AMBG(Σ′, Θ′, [if (d⃗X) e|c⃗X]);
```

**Figure 4.10**. Pseudocode that defines the event propagation AFTER function over the $handle_{switch}$ and $handle_{ambg}$ event handlers.

### 4.3.2 Atomicity

Although the pseudocode in Section 4.3.1 describes the basic algorithm involved in implementing continuous evaluation, an implementation must deal with a couple of atomicity issues. The pseudocode in Section 4.3.1 is not correct when two situations are considered. First, the pseudocode is not correct when another event occurs while another event's changes are being processed. When the latter event causes a condition to become false in a previously active connection where the former event has already caused an earlier condition to become false, the pseudocode can fail to be correct because it can miss detecting that the connection was previously active. Second, the pseudocode is not correct when the same event causes changes in multiple expressions that are contained inside the same circuit. In addition to the problem incurred in the first situation, a circuit can incorrectly become unconnected or ambiguous because event handler processing order is not well defined.

Having another event occur while another event's changes are being processed does not involve multiple threads, which are not supported by SuperGlue. Instead, a change in one signal can trigger changes in other signals, which cause new events to be posted. In SuperGlue code, these changes will involve command streams that are described in Section 4.4. Changes to an import signal with a Java-based driver implementation could also change other signals.

As an example of how the same event can effect multiple expressions in one circuit, consider the following SuperGlue code:

```
if (is_celsius.get)
  view.text = ((3 * thermometer.temperature) / 8) + "C";
else if (!is_celcius.get)
  view.text = thermometer.temperature + "F";
```

In this code, the circuit that implements the **view.**text signal contains two connections whose conditions depend on the value of the **is_celsius.**get signal. When **is_celsius.**get becomes true, the second lower-priority connection becomes inactive while the first higher-priority connection becomes active. If the lower-priority

connection becomes inactive before the higher-priority connection becomes active, the **`view.`**`text` signal becomes incorrectly unconnected.

The solution to atomicity issues in SuperGlue involves serializing state changes and ordering how dispatch of event handlers that are installed on the same Java-based signal driver. First, we must track variable bindings used for individual instances of circuit continuous evaluation. This allows us to uninstall event handlers from conditions without re-evaluating other conditions, which may provide incorrect results when the state of these conditions is changing. Second, we must order the dispatch of event handlers that are installed in the same continuous evaluation context on the same Java-based signal driver. This ordering ensures that conditions and connections are evaluated without spurious errors. Given two event handlers $h_a$ and $h_b$ that are installed in the same continuous evaluation context on the same Java-based signal driver, $h_a$ must be dispatched before $h_b$ if one of the following two criteria is true:

- The expression affected by $h_a$ occurs in a connection of a higher priority than the expression affected by $h_b$; or
- The expressions affected by $h_a$ and $h_b$ occur in the same connections, and the condition affected by $h_a$ occurs later than the condition affected by $h_b$.

Because continuous evaluation can involve multiple circuits, a signal event handler can affect multiple expressions in multiple circuits. As a result, event handler ordering involves traversing a hierarchy of circuits being evaluated. These rules do not work for event handlers that affect expressions in different connections of the same priority. In our prototype, when a connection becomes active or inactive during continuous evaluation, the conditions of all connections of the same priority must be re-evaluated to ensure they are also not becoming true or false. We do not show the pseudocode for this re-evaluation.

We require that an event cannot occur during the processing of another event if it affects the same circuit evaluation. When an event's processing is delayed, the state change that causes the event must also be delayed; otherwise discrete and continuous evaluation will become inconsistent with each other. We must implement this delay in command streams and Java-based signal drivers that change state. We describe how we

deal with streams in Section 4.4.4. For Java-based signal drivers, state changes that affect signals are queued to be processed after the event being processed has occurred.

## 4.4 Stream Layer

As with signals, streams can have circuits that determine how they are evaluated, or can be implemented directly in Java code. The connections in stream circuits are prioritized in the same way as signal circuits. Streams also share some of the evaluation semantics described in Section 4.1 and Section 4.3. The evaluation of a command stream access involves discrete evaluation to determine if the command should be executed. The evaluation of an event stream access involves continuous evaluation to switch what event source is being listened to. Beyond these similarities, stream accesses are very different semantically from signal connections: a stream access is imperative and exposes SuperGlue code to control flow details, while connections are declarative and hide control flow details.

Stream semantics are intertwined with closure semantics. Stream accesses are organized in closures and do not do anything until their containing closures are created. Closures are also related to SuperGlue's object model, where all objects are created in closures and exist as closures.

### 4.4.1 Closures

SuperGlue closures are used in two cases. First, *object closures* are used to express the instantiation context of an object. Second, *stream access closures* are used to capture the context of a handled event or performed command. Both kinds of closures maintain the following lists:

– A closure has a list of objects that are instantiated when the closure is created at run-time. These closures also maintain the circuits for these objects. For this reason, connections expressed in a closure will not affect objects created outside of the closure.

– A closure has a list of command stream accesses that are conditionally executed when the closure is created. Command stream accesses can be guarded by the

same kinds of conditions that guard connections. Each command stream access of a closure is executed only if its conditions are true at the time of the closure's creations.

– A closure has a list of event stream accesses that are conditionally activated after the closure is created. An activated event stream access receives events that are transmitted through the event stream. Like command stream accesses, event stream accesses can also be guarded by the same kinds of conditions that guard connections. An event stream access is activated when its conditions become true after the closure's creation, and is deactivated when its conditions become false after the closure's creation.

Closure creation is a two-step process in SuperGlue. A closure first allocates all of its objects. When an object is allocated, it cannot access other objects that are defined in the closure. For objects with Java driver implementations, object allocation involves calling the driver `create()` method, which returns a Java object to hold the SuperGlue object's identity and state. For objects with SuperGlue implementations, object allocation involves recursively allocating the object's closure. After a closure is allocated, the closure undergoes initialization, which performs the following tasks in order:

– The closure's objects are initialized. During initialization, objects can install event handlers on other objects, even if these objects have not been initialized yet. For objects with Java driver implementations, object initialization involves calling the driver `init()` method with an object that represents the allocated SuperGlue object. For objects with SuperGlue implementations, object initialization involves recursively initializing the object's own closure.

– The closure's command stream accesses are performed if their guarding conditions are true when the closure is created. If a command stream access involves creating a closure, this closure is allocated and initialized before the containing closure's initialization continues.

– The closure's event stream accesses are activated whenever their guarding conditions are true after the closure is created. This involves continuous evaluation of the conditions that guard an event stream access according to the semantics presented in Section 4.3.

Ideally, the order that the objects of a closure are initialized in should not be important. Order is not important for objects with SuperGlue implementations that do not use driver classes that are outside of SuperGlue's core library. Unfortunately, this is not the case for many classes in GlueUI, which depend on Swing class implementations that are sensitive to initialization order. The problem is that many Java objects begin to initialize themselves as event handlers, which can cascade to the point that an object eventually is not able to satisfy a signal request because it has yet to be initialized. Object initialization order in a closure is fixed as object declaration order in a closure definition, so programmers can modify object initialization order as needed.

### 4.4.2 Command Streams

Command stream accesses are only evaluated when their enclosing closures are created. As with a connection, a command stream access can be guarded by multiple conditions. When the enclosing closure of the command stream access is created, these conditions are evaluated to determine if the command stream should be executed. If a command stream is executed, and the command stream is implemented as a circuit, then this circuit is evaluated to determine what other command stream should be executed. If a command stream is implemented with Java code, then the command stream's `execute()` method is called upon its execution. All of this is very similar to the discrete evaluation of a signal, which is described in Section 4.1.

Command streams can update state when they are accessed. If a command stream executes because of an event that also affects the continuous evaluation of a signal, its resulting state changes will not be visible until after continuous evaluation has completely processed the event.

### 4.4.3 Event Streams

As with a connection, an event stream access can be guarded by multiple conditions. After an event stream access's containing closure is created, the access will receive events from the event stream when its guarding conditions are true. This involves installing event handlers on conditions that guard the event stream access to detect if they are all true. When all of its guarding conditions are true, the event stream access is activated so it can receive events. As soon as one or more of its guarding conditions becomes false, the event stream access is deactivated so it no longer receives events. The semantics of this activation and deactivation are very similar to the activation and deactivation of a signal connection in a circuit, which is expressed by the $\texttt{INSTALL}_{\texttt{COND}}$ function in Figure 4.9. Likewise, the evaluation of an event stream circuit is similar to the continuous evaluation of a signal circuit, which is also expressed by the $\texttt{INSTALL}_{\texttt{COND}}$ function in Figure 4.9. If an event stream is implemented as a Java-based driver, then activation and deactivation occurs by directly calling the event stream driver methods.

### 4.4.4 Signals as Streams

Signals are used as event streams with **`on (begin)`** and **`on (end)`** clauses, which transform signal value-change events into event stream events. An **`on (begin)`** clause is an event stream access that detects when its guarding conditions become true. To implement this behavior, an event is dispatched when the event stream access is activated. An **`on (end)`** clause is an event stream access that detects when one of its guarding conditions becomes false. To implement this behavior, an event is dispatched when the event stream access is deactivated.

When a signal is used as an event stream, its change events can be used to both update circuits and execute state-changing command streams. This leads to an atomicity issue: the execution of state-changing command streams can cause additional changes to conditions in the circuits being updated. To ensure that only one event at a time affects a circuit update, our prototype requires that event stream event handlers are not dispatched until after circuit updating event handlers are dispatched.

### 4.4.5 Iterator Streams

Iterator streams can be accessed as either command streams or event streams. Accessing an iterator stream as a command stream is straightforward: the access is executed for each element of the iterator. If an iterator stream is accessed as an event stream, a closure is created for each element that is added to the stream. This closure is different from a normal stream access closure in that it is destroyed when the element it was created for is removed from the iterator stream. When a closure is destroyed, all event stream accesses activated inside the closure are immediately deactivated. Closure destruction will also transmit events to **`on`** (**`end`**) clauses expressed inside the closure.

## 4.5 Performance

Our prototype implementation of SuperGlue is very unsophisticated, and so it does not perform very well. To determine what the performance penalty is for using SuperGlue, we use a series of microbenchmarks. Although these microbenchmarks do not measure the real performance of real programs, they can give us an idea of what programs cannot yet be effectively implemented in SuperGlue.

The first microbenchmark that we consider measures how the performance of a discrete evaluation of a signal compares to the performance of a method call in Java. Calling a method in Java takes between .05 and .06 microseconds on a 867 MHz PowerBook G4 running Java 1.4.2 with Hotspot enabled. We ensure that the method is being accessed virtually by deciding dynamically what test object is created. Accessing the current value of a signal takes between 7.2 and 7.8 microseconds, so a signal access in SuperGlue is about 144 times slower than a method call in Java. The SuperGlue code involved in this benchmark simply makes one signal connection between two objects:

```
let m = new Mock;
let d = new Discrete;
d.signal = m.signal;
```

The second microbenchmark that we consider measures how the performance of a continuous evaluation of a signal compares to the performance of event handling in Java. Receiving a simple event in Java takes between 1.2 and 1.3 microseconds. Receiving

a simple event in SuperGlue takes between 2.6 and 2.9 microseconds, which is only two times as slow as Java. The reason for this is that once an event handler is installed on an object, the two objects can communicate directly through the event handler. As a result, the only extra overhead the SuperGlue code must deal with are extra method calls and some data translation (boxing integers). When considering a circuit with two connections that are constantly switched during continuous evaluation, SuperGlue's performance slows down considerably because the interpreter is always involved in the switching. Receiving an event in Java with conditional switching takes between 1.3 and 1.5 microseconds. Receiving an event in SuperGlue with conditional switching takes around 110 microseconds, meaning that SuperGlue is about 84 times slower than Java in this microbenchmark.

Another major performance problem in SuperGlue is its inability to aggregate the viewing of state embedded in iterator elements. For example, the JavaMail library provides an interface for listening to changes in email messages based on what folder they are in. However, this interface cannot be used in SuperGlue because our driver interface can only provide the value for exactly one email message when an event handler is installed on an email message property. To listen for changes in all messages of an email folder, Java code can install one event handler on the email folder, while SuperGlue code installs event handlers for each message of the email folder. The problem with SuperGlue's approach is that it incurs a **linear** performance penalty that increases with the same of the set being viewed. Solving this performance problem requires re-architecting SuperGlue's driver interface to provide drivers with more flexibility in where they install event handlers. We describe iterator stream enhancements in Section 6.2.

In many user interface programs, SuperGlue code executes rarely because state does not change very rapidly. As an example, in the email client of Section 3.2, SuperGlue code only executes when a mailbox is modified or a user manipulates some widget. In these programs, SuperGlue's performance is acceptable because signals are not being invoked very often. For example, response time in our email client does not appear to be much slower than response time in a typical Swing application. In programs where state changes more often, for example in a simulation of a physical environment, the perfor-

mance of our prototype will not be acceptable. Although constant factor improvements in performance can likely be obtained by tuning our interpreter, compilation techniques will be needed to satisfy the performance requirements of compute-intensive or resource-constrained applications. We describe the viability of SuperGlue code compilation in Section 6.1.

## 4.6 Syntax

Conceptually, SuperGlue can be divided into two languages: a signature language for declaring modules, classes, interfaces, and inner objects, and a statement language for expressing connections and implementing classes. Although these languages are separated for discussion purposes, they are actually used together in the same source files. The rest of this chapter describes SuperGlue's signature syntax (Section 4.6.1), statement syntax (Section 4.6.2), and important syntactic sugar (Section 4.6.3) that makes SuperGlue's use less frustrating.

### 4.6.1 Signature Syntax

The complete syntax of SuperGlue's signature language is shown in Figure 4.11. At the top level, a SuperGlue module is defined in its own file that is named after the module. A module begins by declaring what other modules it will use definitions from (using the **`use`** keyword) and then defines its own classes and interfaces. A class can be defined with the **`native`** keyword, which means it is implemented in Java as a driver, or as **`abstract`**, which means it cannot be instantiated. Next, a class declaration specifies the class's superclass, interfaces, ports, and inner objects. If a class is not native, then it must be implemented with SuperGlue code following the class's declaration, where the class's implementation is preceded by the **`with`** keyword. Interface and inner type declarations are basically the same as was described in Section 2.2.

### 4.6.2 Statement Syntax

The complete syntax of SuperGlue's statement language is shown in Figure 4.12. Statements are defined at the top level to be object instantiations, connections, statement

```
module: module module-id { (use-decl)* (class | interface)* }
use-decl: use module-id;

class: native? abstract? class class-id
       (extends class-id)? (imports intfs)? (exports intfs)?
       { (((import | export) port-decl) | inner-decl)* }
       (with { (use-decl)* statement })?

interface: interface interface-id
           (extends interface-id (, interface-id)*)?
           { (port port-decl)* }

port-decl  : (event | command | iterator)? port-id : port-type
port-type  : primitive | void | class-id |
             interface-id | inner-id

inner-decl: inner inner-id (extends facaide-id)?
            (imports intfs)? (exports intfs)? { (port-decl)* }
intfs: interface-id (, intfs)?
```

**Figure 4.11**. The syntax of SuperGlue's signature language; the * and ? regular-expression operators respectively express zero or many repetition and zero or one optionality.

blocks, variable declarations, and stream accesses. Connections are to ports (left-hand side) and from either other ports (for stream accesses) or expressions (for signals).

### 4.6.3 Syntactic Sugar

SuperGlue has many forms of syntactic sugar that are designed to decrease verbosity in SuperGlue code. First, simple unguarded connections can be expressed when an object is created in a constructor-like way. For example, consider the task of configuring a **timer** object to tick every 500 milliseconds. Without syntactic sugar, the **timer**'s `tick` signal is connected and **timer** is instantiated in separate statements:

```
let timer = new Timer;
timer.tick = 500;
timer.tick = 2;
```

```
statement : new | connection | if | block | connect-var |
            let | on | do | for
new       : let object-id = new class-id;
connection: port-ref = (expression | port-ref);
port-ref  : object-var . port-id
            (( expression (, expression)* ))?
if        : if ( condition ) statement |
            if ( condition ) statement else statement
condition : expression | query | condition && condition
expression: port-ref | object-id | var-id |
            constant | binary | ...
binary    : expression bin-op expression | ...
bin-op    : < | > | == | || | && | ...
block     : { statement* }

connect-var: var ( var-id : var-type ) statement
var-type   : class-id | object-var . inner-id |
             interface-id | primitive
object-var : var-id | object-id
query      : var-id = < var-type > expression

let: let var-id = expression;
on : on ( port-ref ) statement |
     on ( var-id = port-ref ) statement
do : do   port-ref ; |
     do ( port-ref ) statement |
     do ( var-id = port-ref ) statement
for: for ( var-id = port-ref ) statement
```

**Figure 4.12**. The syntax of SuperGlue's statement language.

Writing down two statements is overly verbose because `tick` and `mode` never change. Instead, we can connect `tick` and `mode` in the **`new`** statement using constructor-like syntax:

```
let timer = new Timer(tick = 500, mode = 2);
```

Constructor-like syntax allows simple unguarded connections to be expressed in object instantiations with less typing because the target object is not specified. Additionally, more complicated guarded connections are more prominent because they are not mixed with numerous simple connections.

In Section 4.2.6, we mentioned that connections can be specified in modules in order to express default behavior. Additional syntactic sugar eases the expression of default connections by allowing them to be expressed in signal declarations. For example, consider the rows of a user-interface table view, which are connected to an empty list by default. Without syntactic sugar, the rows declaration and connection of an empty list to table rows are expressed separately:

```
class TableView extends CompoundEditor {
  import rows : List<Row>;
  ...
}
var (table : TableView) table.rows = [];
```

With syntactic sugar, the signal declaration can be combined with the default connection:

```
class TableView extends CompoundEditor {
  import rows : List<Row> = [];
  ...
}
```

Implicitly, a connection variable is defined in each class that is typed by that class. When a default connection is expressed to a port, it is simply transformed into a connection that targets this connection variable.

Our final form of syntactic sugar eases the expression of connections to ports declared in interface and inner types. Without syntactic sugar, expressing a connection to a port that is contained in a port requires two things: the former port must have a unique inner object (i.e., the inner type used by any other port), and a connection variable must be used to abstract over this unique inner type. For example, consider declaring the insets for rows and columns of a table view:

```
interface Dim<T> extends List<T> {
  import insets : int;
}
class TableView extends CompoundEditor {
  inner Rows    imports Dim {}
  inner Columns imports Dim {}

  import rows    : Rows;
  import columns : Columns;
```

```
  ...
}
```

Although rows and columns have the same structure, they have different types so their insets can be connected individually. Connecting insets for rows and columns then requires the use of two connection variables:

```
let table = new TableView;
var (rows    : table.Rows   ) rows .insets = 3;
var (columns : table.Columns) columns.insets = 5;
```

This code is overly verbose for two reasons. First, `TableView` must define two inner types that do not add any additional ports. Second, two inner object variables must be declared for each connection even though they are both used to abstract over exactly one value. Syntactic sugar in SuperGlue implicitly defines a unique inner type for each port that is typed by an interface or inner object. This type is then used to implicitly define a connection variable for any connection to a port that is accessed through this port. As a result, we do not need to explicitly declare unique inner types for the rows and columns of a table view:

```
class TableView extends CompoundEditor {
  import rows    : Dim;
  import columns : Dim;
  ...
}
```

We can then directly connect the insets for rows and columns through port accesses:

```
table.rows.insets = 3;
table.columns.insets = 3;
```

This kind of syntactic sugar allows inner objects and interfaces to be used without the verbosity of additional types and connection variables.

# CHAPTER 5

# RELATED WORK

SuperGlue is closely related to languages in the following five areas:

– **Functional-reactive programming languages**: SuperGlue's signal abstractions are similar to signal abstractions in functional-reactive programming languages. We compare SuperGlue to these languages in Section 5.1.

– **Object-oriented programming languages**: SuperGlue's object-oriented abstractions are similar to dispatch and inheritance mechanisms in object-oriented languages. We compare SuperGlue to these languages in Section 5.2.

– **Logic programming languages**: SuperGlue's use of rules to connect signals is similar to how rules are used in logic programming languages. We compare SuperGlue to these languages in Section 5.3.

– **Constraint-imperative programming languages**: SuperGlue's signal connections are similar to simple constraint expressions that are supported in constraint-imperative programming languages. We compare SuperGlue to these languages in Section 5.4.

– **Component programming languages**: SuperGlue is based on a port and connection paradigm, which is the basis for many other kinds of component programming languages. We compare SuperGlue to these languages in Section 5.5.

## 5.1 Functional-reactive Programming

Functional-reactive programming (FRP) systems integrate continuous and discrete abstractions for viewing state into a functional programming language. The idea behind

FRP is that functions can be "lifted" to transform signals. For example, if `X` is a signal and `f()` is a function, then `f(X)` changes as `X`'s value changes. We compare SuperGlue to the following three very different FRP systems:

- **Yampa** [20] is the most recent Haskell-based FRP system. Yampa is synchronous, which means continuous evaluation of signals occurs according to a clock with an explicit representation in the program. For modularity and efficiency reasons, signals in Yampa are second-class values, which means they cannot be directly organized into graph-like structures. As a result, Yampa supports special switching and collection abstractions to deal with the resulting connection scalability problem. SuperGlue is compared to Yampa in Section 5.1.1.

- **FatherTime** [6] (FrTime) is a Scheme-based FRP system. Unlike Yampa, FrTime is not synchronous. Also, unlike Yampa, signals in FrTime are first-class values. As a result, FrTime does not suffer from connection scalability problems, and does not require special abstractions to organize signals into graph-like structures. SuperGlue is compared to FrTime in Section 5.1.2.

- **Frappé** [7] is an implementation of FRP for the Java programming language. Frappé's unique feature is that signals are integrated with the JavaBeans [38] event and property model. SuperGlue is compared to Frappé in Section 5.1.3.

### 5.1.1 Yampa

Yampa is the most recent of many Haskell-based FRP systems that began with the Fran animation system [13]. Yampa, Fran, and their variations have been shown to be useful in domains such as animation [13], user interfaces [8, 12, 35], video games [9], and robotics [20, 33].

All Haskell-based FRP systems are synchronous because the goal of these systems is to preserve the purity of the functional programming paradigm, which requires that state and time be modeled explicitly. Synchronous evaluation has many advantages: programs are easier to reason about, support general time-based transformations, and cannot exhibit race conditions. Also, synchronous evaluation is more suitable for interactive

programs that require accurate time-based numerical computations such as integration. However, synchronous code cannot interact very easily with imperative code or any state outside of the program that is not synchronous. For this reason, SuperGlue is not synchronous: synchronization is the responsibility of object implementations and glue code, and time-based numerical computations can only be approximated.

Fran, an early Haskell-based FRP system that preceded Yampa, supported the use of signals as first-class Haskell values. However, the use of signals as first-class values in Fran leads to *space-time leaks* [11], which occur when an unbounded amount of time-based delayed computation needs to be caught up at some point in a program's execution. As a result, in Yampa, signals are second-class values and only signal functions, which cannot be curried, are first-class values. SuperGlue signals are also second-class values, but for a different reason: the true identity of an object that contains a signal must be known when organizing the signal's connections in a circuit. Signals in SuperGlue can only be connected through objects or connection variables, which is analogous to the second-class status of signals in Yampa.

Because of the second-class status of signals in Yampa and objects in SuperGlue, state in these languages cannot be organized directly into graph-like structures such as trees or lists. As described in Section 2.2, SuperGlue deals with state in graph-like structures through object-oriented abstractions such as inner objects and connection variables. Yampa deals with state in graph-like structures through dynamic collection abstractions [30]. Dynamic collections are collections of stateful signal functions that support the following operations:

– An input signal can be broadcast to all the signal functions of a collection. Alternatively, a routing function can be specified that determines what input signal is delivered to each signal function of a collection. This feature compares to connection variables in SuperGlue, which can connect multiple objects and inner objects of a specified type. Routing functions are similar to conditions that guard connections through connection variables.

– Signal functions can be added to or removed from a collection on the direction of discrete events without disturbing the state of other signal functions in the collection. In SuperGlue, collection updates are encapsulated inside objects with imperative implementations. A generic mutable collection can be expressed as an object of the `ListCell` class in SuperGlue's core library.

Directly comparing SuperGlue's object-oriented abstractions to Yampa's dynamic collection abstractions is difficult because of their different design goals. SuperGlue's object-oriented abstractions are designed to abstract over multiple signal connections that are used in Java-based object implementations. For example, glue code can abstractly specify how node inner objects of a GlueUI tree view are connected, but the Java implementation of a tree view must create the node inner objects for these connections to be useful. In other words, SuperGlue programs depend on a lot of "magic" in the form of Java-based object implementations. On the other hand, Yampa's dynamic collections are designed to maintain purity of the functional programming paradigm. For this reason, SuperGlue is a more practical but less theoretically grounded language than Yampa.

### 5.1.2 FatherTime

FatherTime (FrTime) is a Scheme-based FRP system that differs significantly from Haskell-based FRP systems like Yampa. Unlike Yampa and similar to SuperGlue, FrTime is not synchronous. As a result, FrTime code can integrate with imperative Scheme code in a way that is similar to how SuperGlue code integrates with Java code. Unlike Yampa and SuperGlue, signals in FrTime are first-class values, which means they can be used like normal Scheme values in Scheme code. Because the full power of Scheme can be used to manipulate signals, graph-like state can be expressed and used directly in FrTime code. As a result, unlike SuperGlue and Yampa, FrTime does not require any special abstractions to deal with graph-like state. For example, a tree of nodes can be expressed by a higher-order function that represents a tree node and provides access to the higher-order functions that represents the node's children. As a result, an email client's folder view tree described in Section 1.1 could be defined in FrTime code as follows:

```
(define folder-view
  (new-tree-view mailboxes
    (lambda (node)
      (if (isa-mailbox node)
        (get-root-folders node)
        (get-sub-folders node)))))
```

In this code, a higher-order function is used to configure the nodes of a tree view. Higher-order functions in FrTime replace the functionality of SuperGlue's connection variables in abstracting over connections. Besides using a higher-order function rather than a connection variable, this code is similar in structure and complexity to the SuperGlue code that defines the same folder view tree in Section 2.2.

Conditions in FrTime can refer to values that are signals. As a result, conditions that depend on state can be expressed directly in FrTime, as they can be in SuperGlue. For example, the following FrTime code displays as rows in a message view table the email messages of a folder under the conditions that only one node is selected in the folder view tree and that node is a folder:

```
(define message-view
  (new-table-view
    (if (and (== 1 (size (selected folder-view)))
             (isa-folder (get (selected folder-view) 0)))
      (get-messages (get (selected folder-view) 0))
      (empty-list)) ...))
```

As in SuperGlue, the conditions of this FrTime code are re-evaluated as selection in the folder view tree changes. As with the definition of the folder view tree, this FrTime code is similar in structure and complexity to SuperGlue code that defines a message view tree in Section 2.2.

FrTime is clearly more powerful than SuperGlue because FrTime code can leverage the full power of recursion and higher-order functions when manipulating signals. However, it is not clear how this power translates into useful expressiveness when gluing together components in interactive programs. The FrTime examples in [6] do not make use of recursion and only use higher-order functions to iterate or abstract over data.

Because SuperGlue's abstractions focus on expressing signal connections between objects, its design decisions differ from FrTime's. In particular, connections in SuperGlue are expressed through rules, where signal connections to the same object do not need to be expressed in the same file. In FrTime, an object's fields must be initialized to signals when it is created, meaning that all of the object's signal connections are expressed at this time. Although FrTime is powerful enough to support rules that connect its objects (via a rule interpreter encoded in Scheme), the use of rules would change the paradigm in which FrTime programs are written.

### 5.1.3 Frappé

Frappé is an implementation of FRP in Java. Code in Frappé can integrate with Java code in the following ways:

- Any JavaBeans component can receive or provide signals.
- The JavaBeans event model is used to propagate signal changes.

With these two integration features, JavaBeans components can be connected together using signals into assemblies, which can be reused as JavaBeans components.

JavaBeans' bounded properties play a critical role in Frappé. As discussed in Section 3.1, properties describe state that can be mutated by multiple entities. Bounded properties in JavaBeans are properties with methods for accessing their current values, for listening to changes in their values, and for changing their values. Frappé can automatically convert bounded properties into signals. A signal can then be used in "lifted" methods calls, which call a Java method whenever the signal's value changes.

Frappé and SuperGlue have a similar goal: to utilize signals to improve how Java components are glued together. Frappé can be easier to use than SuperGlue because signals are easy to create from existing Java code. Converting bounded properties in SuperGlue involves writing more adapter code than in Frappé. However, we could provide special support for bounded properties in the same way that Frappé does. The primary problem with Frappé is that a lot of interesting signals in existing Java code are not properties. As discussed in Section 3.1, these signals can be configuration options

(e.g., foreground color) or internally managed state (e.g., current selection). SuperGlue provides a more general way than Frappé of using Java code through signals, which is more flexible but can be more difficult to use.

Frappé lacks a separate syntax for using signals; instead signals are used through a special object-based Java API. As a result, signal programming in Frappé is very verbose. As future work, the author talks about designing a separate Fran-like [13] language for using signals in Frappé [7]. This separate language approach is already used in SuperGlue. The problem with their proposed approach is that it would enforce a Haskell-like programming model to glue together Java code. When compared to SuperGlue, their proposed approach would result in a language that is unfriendly to Java programmers, and could not easily represent Java class hierarchies to glue code.

## 5.2 Object-oriented Programming

SuperGlue supports graph-like state with object-oriented abstractions. We are currently unaware of any object-oriented language that supports something similar to SuperGlue's inner object abstraction. The problem domain of SuperGlue's inner object abstraction is similar in scope to the problem domains of the fly-weight, facade, visitor, and adapter design patterns [18], which are commonly used in object-oriented programs. Inner objects enable new signals to be related to a value: an inner object that a value is connected to conceptually wraps the value with new signals. As a result, inner objects can enhance the functionality of an existing value through new signals. The use of inner object abstractions in this way makes them similar to mixin [3] or open class [5] abstractions, which are used in object-oriented languages to add new methods to existing classes. The type of a value that is connected to an inner object can also act as a secondary prioritization mechanism. The use of a secondary type in prioritizing connections is analogous to how calls to multimethods [4] can be dispatched according to argument types.

Arbitrary conditions in SuperGlue can guard the applicability of signal connections, which is analogous to how conditions can guard method dispatch in languages that support predicate dispatch [14, 29]. However, SuperGlue does not perform any static

checking to ensure exactly one connection is usable at any given time, where such checking is common for methods in object-oriented languages that support predicate dispatch. Static checking is not possible in SuperGlue because conditions are not simple arithmetic expressions and can refer to signals whose implementations are encapsulated. Additionally, the use of a signal being connected inside an object implementation may also be guarded by other conditions that are encapsulated inside the object implementation. For this reason, run-time checking of ambiguous signal connections is used in our current implementation. As future work, Section 6.2 proposes enhancing SuperGlue with a type system that would enable the static and modular detection of signal connection errors.

## 5.3 Logic Programming

SuperGlue code encodes signal connections through rules that are similar to rules in logic programming languages such as Prolog [36], which are based on mathematical logic. Similar to Prolog, signal evaluation in SuperGlue involves backward chaining: evaluation of one signal can involve evaluating other signals referred to in conditions that guard the former signal's connections. Unlike Prolog, the evaluation of rules in SuperGlue are deterministic. However, SuperGlue's support for continuous evaluation relies on something like backtracking to reconnect signals as conditions that guard connections change their values. SuperGlue also does not support anything similar to variable unification.

With signals, SuperGlue supports the concepts of state and time in a purely declarative way. Many logic programming languages also provide support for reasoning about state and time; see [32] for a comprehensive survey. In temporal logic languages, the validity of a fact can be parameterized with a time range. Unlike SuperGlue, time is modeled explicitly in temporal logic languages, which limits how these languages can be used in interactive programs.

Kernel Prolog [41] supports a *fluent* abstraction, which are stateful objects that are represented as infinite "sources of answers" to Prolog rule evaluations. Similar to Super-

Glue's signals, fluents can be composed and can be implemented in Java code. Unlike SuperGlue signals, fluents do not support continuous evaluation.

## 5.4 Constraint-imperative Programming

Constraint variables have values that are determined by what constraints are placed on them. When compared to SuperGlue, constraint variables are similar to signals and constraints are similar to connections between signals. However, constraints are more powerful than signals: a constraint variable is multidirectional, which means that it is not restricted to being an output or input, and a constraint variable can be placed under multiple constraints at the same time. In contrast, a signal is unidirectional and must always be specified as either an import or export.

Constraint imperative programming (CIP) languages [16, 17] support constraint variables whose constraints can change during program execution via imperative updates. As a result, the value of a constraint variable can change dynamically, which is similar in behavior to a signal. CIP languages also support the control flow and mutable variable constructs needed to perform general-purpose programming tasks. The Kaleidoscope language [16, 17] supports CIP with object-oriented abstractions. In particular, Kaleidoscope can be used to create constraints over complex user-defined objects, which is similar to connecting signals in SuperGlue-implemented objects.

Although constraints are more expressive than SuperGlue connections, Kaleidoscope programs often create constraints that are used in connection-like ways: each of these constraints is permanent and is the sole constraint of a constraint variable. As an example, consider the example Kaleidoscope code in Figure 5.1 from [23], which implements a simple user interface. In this code, the **`always`** keyword is used to create constraints that hold forever, while the **`assert`** keyword creates a constraint that holds for the duration of its enclosing **`while`** loop. SuperGlue code equivalent to the code in Figure 5.1 is shown in Figure 5.2. The SuperGlue and Kaleidoscope code are almost the same. The problem with CIP is that the additional power of a constraint over a connection, which is gained through a constraint solver, is not often useful, while the extra power can also make constraints more difficult to use. In particular, programmers must explicitly

```
always: temperature = mercury.height / scale;
always: white_triangle.top = thermometer.top;
always: white_triangle.bottom = mercury.top;
always: mercury.bottom = thermometer.bottom;
while (mouse.button = down)
  assert mercury.top = mouse.location.y;
end while;
```

**Figure 5.1**. Kaleidoscope code that allows a user to drag the mercury of a thermometer up and down with a mouse.

```
let temperature = mercury.height / scale;
white_triangle.top = thermometer.top;
white_triangle.bottom = mercury.top;
mercury.bottom = thermometer.bottom;
if (mouse.button == down)
  mercury.top = mouse.location.y;
```

**Figure 5.2**. SuperGlue code that is equivalent to the Kaleidoscope code in Figure 5.1.

manage the duration of the constraint (**always**, **once**, or **assert**), and programmers must explicitly manage the constraint's direction, which determines what variables can be updated by the constraint.

Kaleidoscope can be useful in numerical domains where multiway constraints are useful. Layout managers in user-interface libraries often involve multiway constraints, which can directly be expressed with Kaleidoscope code but not with SuperGlue code. However, user-interface program do not involve many kinds of these constraints. Therefore, SuperGlue connections are often expressive enough for glue code tasks while being easier to use than Kaleidoscope's constraints.

Through general purpose control-flow constructs, Kaleidoscope supports imperative programming in a much more seamless way than SuperGlue. Although the mostly free mixing of imperative and declarative code complicates Kaleidoscope's semantics and implementation, programmers are mostly able treat Kaleidoscope as a standard object-oriented language with constraint programming extensions. Compared to Kaleidoscope, SuperGlue's support for imperative programming is not as fluid because streams can-

not directly influence port connections, and programmers cannot view SuperGlue as a conventional object-oriented language with connection and signal extensions. For this reason, SuperGlue does not support imperative programming as nicely as Kaleidoscope, but neither does it suffer from the complexity of mixing imperative and declarative abstractions.

## 5.5 Component Programming

As mentioned in Section 1.3, the port-connection paradigm is often used in component (and module) systems because of its support for component dependencies that are explicit and configurable. In contrast to connections, dependencies that are expressed with more powerful procedure calls are more difficult to reason about and change. The drawback of the port-connection paradigm is that connections are not very expressive, and are often only good at representing coarse-grained or abstract (conceptual) dependencies.

Module systems are well-suited to connections because module dependencies are usually of a coarse grain. For example, connections were very effective in Jiazzi [27], which was a Java module system designed by this author and is based on program units [15]. Jiazzi supports module dependencies that are explicitly specified and configured through connections. Connection expressibility problems are not a big issue in Jiazzi because connections are specified between coarse-grained packages of classes.

Besides being used in module systems, connections are often used to describe dependencies between components in software architectures. One such language is ArchJava [2], where components have ports that are explicitly connected together in glue code. Type-correct ArchJava programs adhere to the property of *communication integrity* [24], which ensures that intercomponent communication conforms to the program's specified architecture. With alias annotations, ArchJava can enforce communication integrity even when components communicate through shared objects. Similar to ArchJava, SuperGlue also enforces communication integrity, but does not track communication through shared objects. ArchJava and SuperGlue differ in how they support dynamic architectures, where connections can change during run-time. ArchJava allows connections to be

created at run-time as long as they conform to statically verified connection patterns. In contrast to ArchJava, SuperGlue supports dynamic architectures through circuits, which can switch how a port is connected dynamically; closures, which enable the creation of new objects with a specification of their connections; and connection variables, which allow connections to be specified without targeting specific objects and inner objects.

# CHAPTER 6

# CONCLUSION

SuperGlue combines signal, object, and rule abstractions into a novel language for building programs out of state-processing components. SuperGlue improves on established languages, such as Java, with signals that ease how state-processing components are glued together. When compared to other languages that support signals, SuperGlue is unique in its use of object-oriented abstractions to deal with graph-like state. SuperGlue makes a different set of tradeoffs from these languages. SuperGlue does not support abstractions such as recursive higher-order functions; instead, it supports object-oriented abstractions that are easier to use but less expressive. Despite being less expressive, SuperGlue can still accommodate many component assembly tasks.

Section 3.2 shows that SuperGlue can reduce by half the number of operations needed to implement a realistic user-interface program. We showed that when dealing with user-interface features that process state, SuperGlue's use is very beneficial when compared to Java. Only a small number of features in most user-interface programs process state: the other features only react to user input, which does not involve processing state. As a result, SuperGlue can benefit the implementations of only a few features in most user-interface programs, although the dominating complexity of how these features are implemented in existing languages can still make SuperGlue's use worthwhile. Section 6.3 describes why state-processing components can be used more often in user-interface programs, where SuperGlue's use is more worthwhile in these programs.

Beyond building user interfaces, SuperGlue can also be used in implementing tasks that traditionally process data as state-processing programs. For example, Section 3.3 demonstrates how SuperGlue can be used to implement a language-aware editor with state-processing parsing and type-checking components. Building state-processing com-

ponents is naturally more difficult than building data-processing components. However, once state-processing components are built in SuperGlue, gluing them together is similar in complexity to gluing together data processing components.

SuperGlue can be improved in many ways. First, SuperGlue's implementation and libraries need to be polished so that it can be released and undergo user testing. Otherwise, SuperGlue can also be improved in three areas:

- SuperGlue's run-time can be improved to support better performance, more flexibility, and enhanced development tool support (e.g., debugging). This work is described in Section 6.1.

- SuperGlue's language abstractions can be evolved and improved to make SuperGlue more complete, easier to use, and more expressive. This work is described in Section 6.2.

- New SuperGlue libraries can be built to expand and enhance the kinds of programs that can be easily implemented in SuperGlue. In particular, we can look at new kinds of programs whose state-processing implementations are too complicated in existing languages. This work is discussed in Section 6.3.

## 6.1 Technology Enhancements

Our current prototype implementation of SuperGlue is unsophisticated. Code is evaluated with a recursion-based interpreter that is inefficient even by interpretation standards. The performance of our prototype implementation is suitable for user-interface programs, where glue code does not execute very often. However, if SuperGlue is to be used in compute-intensive or resource-limited programs, it will need an implementation with better performance.

The easiest way to improve SuperGlue's performance is through compilation: circuit evaluation currently involves a lot of work that does not change over time. Given a circuit with a static structure, generating code to implement discrete and continuous state viewing is only a matter of generating numerous condition statements. The difficult part of supporting code generation in SuperGlue is in the implementation of drivers: instead

of simply adapting between SuperGlue and Java code, each driver must be implemented as a code generator. To make drivers easier to implement as code generators, we must develop a SuperGlue extension that allows quoted Java code to be expressed with Java code.

Besides exploring a faster implementation, we can explore how connections can be added to or removed from a circuit during program execution, which can facilitate interactive development and dynamic program updates. Changing a circuit at run-time can allow a programmer to quickly try out new ideas or fix bugs. Supporting dynamic circuit modifications with a language-aware editor of the kind as described in Section 3.3 can form the basis of a very powerful development environment for SuperGlue.

Development environments should also support the debugging of SuperGlue code. Because SuperGlue is a declarative rule-based language, conventional debugger breakpoints cannot be used in SuperGlue. Instead, more data-centric watchpoints would be used instead to monitor changes in signal values. A development environment must also support the debugging of Java code and SuperGlue code at the same time, because bugs may exist in driver implementations. For this reason, we plan to explore how a SuperGlue debugger can be built on top of Eclipse's Java debugger [21].

## 6.2 Language Enhancements

Although SuperGlue is complete enough to be used in the construction of realistic programs, it still lacks polishing features needed for realistic software development. First, SuperGlue lacks a very strong static type system, and many kinds of assembly errors cannot be detected until run-time. Also, as mentioned in Section 2.5, SuperGlue currently lacks error handling and multithreading abstractions. Finally, as mentioned in Section 2.3 and Section 4.5, SuperGlue's support for aggregate data cannot take advantage of signal-based code reduction and adversely affect SuperGlue's scalability with respect to performance. The rest of this section describes strategies for resolving these issues.

### 6.2.1 Static Type Checking

SuperGlue's current support for static type checking is very unsophisticated: signal existence is checked by the static type of a containing expression, and signal connections are checked to see if primitive and class types match. Two kinds of type errors can occur in SuperGlue program that are only detected at run-time:

- A signal can be unconnected or connected ambiguously when used. These errors usually occur because a condition guarding a connection is false when the programmer expected it to be true, or because independently developed modules unknowingly connect the same signals using similar types.

- A programmer expresses the wrong type in a connection query, causing the query to always be false as a condition. Mistyped connection queries do not directly cause run-time errors. In the best case, mistyped connection queries lead to unconnected signal errors because the connections they guard as conditions cannot occur. In the worst case, mistyped connection queries cause lower priority connections to be used, forcing the programmer to wonder why their program is not behaving correctly.

A traditional static type system could not detect these errors without severely limiting expressiveness. Instead, we could rely on a general-purpose theorem prover or bug checking system to detect cases where signals are connected in bad ways. Because signal accesses are so powerful, these systems would probably be unsound, meaning they might not be able to detect all problems. In this case, dynamic type checking would still be used to ensure safety at run-time.

### 6.2.2 Error Handling

SuperGlue currently has no special abstractions for detecting or recovering from built-in or user-defined signal errors. It is debatable whether special abstractions are actually needed. For example, failures that are not due to programming errors, which involve error handling rather than debugging, often occur because of resource failures. The status of a resource can be indicated by a another signal, which can guard access to

the resource and can be used to handle conditions where the resource has failed. To make this approach more convenient, an "exception" signal can be associated with normal signals through special syntax that resembles **`throws`** clauses in Java. For example, a `read` signal can be declared to embed an `IOException` signal that communicates the `read` signal's current input-output status. Static type checking can also ensure that the use of a signal is always guarded by its embedded exception signal.

### 6.2.3 Multithreading

It is sometimes necessary for the SuperGlue code of a program to execute in multiple threads. For example, a long running operation in a user-interface should not execute in the user-interface thread. In this case, SuperGlue component assemblies that are executing in different threads could communicate through messaging mechanisms that are not very high-performance but can easily be automated in the run-time. Supporting high-performance multithreading, where threads communicate through shared memory rather than messages, in SuperGlue is not currently a priority because SuperGlue is not currently a high performance language. Also, SuperGlue's support for state communication through signals conflicts with how state is communicated through shared memory. When state is viewed through a signal, the viewer can automatically be notified of changes in the state, so it always has a consistent view of the state. On the other hand, shared memory depends on various forms of locking to ensure state is not modified while it is being viewed. It is not obvious why or how both of these schemes would work together.

### 6.2.4 Iterators

Iterator streams as described in Section 2.3.3 can only be used as event or command streams: they cannot be used in conditions like signals. SuperGlue can currently only communicate values one at a time, and iterators represent an aggregate of multiple values. To use aggregate values in SuperGlue, they must first be split up into multiple single value contexts. This split requires programmers to deal with control-flow details, and has serious performance problems. For example, viewing a property in every element of

an iterator requires installing a separate event handler on each element, even if property changes are communicated for each element from a single event source.

We plan to replace iterator streams with *iterator signals* that can be accessed as signals rather than as command and event streams. Doing this would require allowing one variable to represent an aggregate of multiple values that can be filtered directly in **if** statements. For example, the following code could filter out all messages that are deleted in a message view:

```
for (msg = folder.messages.all)
  if (!msg.deleted.get)
    messageView.rows = new IterationToList(input = msg);
```

The `msg` variable is bound to all of the messages that are contained in an email folder. These messages would then be filtered by a condition that guards against deleted email messages. Finally, remaining undeleted messages would be connected to the `input` signal of a `IterationToList` object, which transforms the messages from an iteration into a list that can be viewed as rows in a user-interface table. The use of an iterator signal in this example allows the filtering of deleted messages to be expressed in a very concise way. The code in this example can also performs adequately when the targeted email folder has thousands of messages. In this case, the detection of when a message is deleted or undeleted involves installing only one event handler on the email folder rather than a separate event handler for each email message contained in the folder.

We have not added iterator signals to SuperGlue yet because our prototype requires substantial changes to handle variables that can possibly be bound to multiple values. More significantly, SuperGlue's driver model must be changed to take advantage of iterator signals by supporting event handlers that listen to changes in more than one element. These changes will add substantial complexity to the driver model, and so we must think them through very carefully.

## 6.3 Applications

SuperGlue's success as a programming language depends on what new kinds of programs it enables. Beyond conventional user-interface programs, we envision Super-

Glue being useful in the development of software agents, which autonomously organize, integrate, and react to changing data sources on behalf of a user. Agents are especially important in ubiquitous computing [44], which focuses on how software can become invisible to users that benefit from them. We also envision SuperGlue being used to build user-interfaces that use state-processing components so they are more interactive than current user-interface programs.

### 6.3.1 Ubiquitous Computing

Agents are analogous to robots that operate in information-centric rather than physical environments. Currently, the most popular kinds of agents operate on web-based data that change infrequently. Such agents cannot benefit significantly from SuperGlue implementations because they process immutable data and not state. However, information that changes often is becoming increasingly available through technologies such as wireless networks, cheap sensors, global position systems (GPS), and radio frequency identifiers. In ubiquitous computing, agents automatically and continuously process and reason about this information on behalf of a user. Because such agents process state, they would benefit significantly from SuperGlue implementations.

As an example of a ubiquitous computing scenario, an agent could notify a user when a desired public bus is predicted to arrive at some location within the time it takes for the user to walk to that location. In such a scenario, the locations of the user and buses would be tracked using GPS, and traffic conditions would be tracked using cameras. Locations and traffic conditions would be transmitted at regular intervals to the agent over a wireless network. The agent would use the information to select a best bus, which would get the user to a desired location within a certain time frame. As the user moves, the buses move, and traffic conditions change, the agent continuously computes what the best bus is. Finally, when the best bus approaches walking range of the user's current location, the agent would alert the user to start walking to a certain bus stop.

Implementing a ubiquitous computing agent involves the following tasks, which would significantly benefit from SuperGlue implementations:

- **Filtering:** because network and computation resources are limited, an agent must avoid receiving and processing unneeded information. In our example, the agent only needs to receive over a wireless network information about buses that are going to a certain location, and can possibly come within walking distance of a user. In SuperGlue, filtering is expressed through conditions that guard what state is viewed by a program. Limiting what state is viewed by a program saves resources because a program, as a client, does not need to receive the state over a network.

- **Integration:** an agent must integrate information from unrelated sources. In our example, the agent must extrapolate the future position of a bus at some time using the bus's current position and traffic conditions. SuperGlue eases how integration is implemented through signal expressions that also behave like signals. For example, the SuperGlue expression that predicts a bus's current position is a signal whose value is automatically recomputed whenever the bus's position or traffic conditions change.

- **Notification:** an agent must detect events that require actions to be performed. In our example, the agent must detect when the best bus is close enough to a user. SuperGlue eases how notification is implemented with stream operations (**`on`** (**`begin`**) in Section 2.3) can detect when signal-referencing conditions become true or false.

Supporting ubiquitous computing in SuperGlue requires building new libraries that support the collection, aggregation, and communication of sensor data. Other more domain-specific libraries are needed to process sensor data in specific applications, e.g., libraries are needed in our example to extrapolate bus positions.

### 6.3.2 Improving User Interfaces

Many user-interface programs are not very interactive because they heavily depend on imperative, or modal, forms of user input and output. For example, many enterprise user-interface programs are currently implemented as web-based user interfaces. Al-

though web-based programs are easier than desktop programs to develop and deploy, a user's experience suffers because web pages are less interactive than desktop widgets. The designers of the Morphic user-interface toolkit refers to support for interactive behavior as *liveness* [26], which is described by John Maloney [25]:

> Morphic is inspired by another property of the physical world: *liveness*. Many objects in the physical world are active: clocks tick, traffic lights change, phones ring. Similarly, in Morphic any morph can have a life of its own: object inspectors update, piano rolls scroll, blobs crawl around. Just as in the real world, morphs continue to run while the user does other things. In stark contrast to user interfaces that wait passively for the next user action, Morphic becomes an equal partner in what happens on the screen.

With a user-interface library that is more like Morphic than Swing in its design philosophy, SuperGlue could be used to build user-interface programs that are highly interactive. When components in this library are combined with state-processing networking components, SuperGlue could become a very nice platform for programming interactive network-based programs.

# REFERENCES


[1] M. Abernethy. Ease Swing development with the TableModel-free framework. http://www-128.ibm.com/developerworks/xml/library/j-tabmod/.

[2] J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in ArchJava. In *Proceedings of European Conference on Object-oriented Programming (ECOOP)*, volume 2374 of *Lecture Notes in Computer Science*, pages 334–367. Springer, 2002.

[3] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of Object-oriented Programming Systems, Languages, and Applications (OOPSLA) and European Conference on Object-oriented Programming (ECOOP)*, volume 25 (10) of *SIGPLAN Notices*, pages 303–311. ACM, 1990.

[4] C. Chambers. Object-oriented multi-methods in Cecil. In *Proceedings of European Conference on Object-oriented Programming (ECOOP)*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56. Springer, 1992.

[5] C. Clifton, G. T. Leavens, C. Chambers, and T. D. Millstein. Multijava: Modular open classes and symmetric multiple dispatch for Java. In *Proceedings of Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 35 (10) of *SIGPLAN Notices*, pages 130–145. ACM, 2000.

[6] G. Cooper and S. Krishnamurthi. FrTime: Functional reactive programming in PLT Scheme. Technical report, Brown University, Apr. 2004. Technical Report CS-03-20.

[7] A. Courtney. Frappé: Functional reactive programming in Java. In *Proceedings of Practical Applications of Declarative Languages (PADL)*, volume 1990 of *Lecture Notes in Computer Science*, pages 29–44. Springer, 2001.

[8] A. Courtney. Functionally modeled user interfaces. In *International Workshop on Design, Specification and Verification of Interactive Systems*, volume 2844 of *Lecture Notes in Computer Science*, pages 107–123. Springer, 2003.

[9] A. Courtney, H. Nilsson, and J. Peterson. The Yampa arcade. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, pages 7–18. ACM, 2003.

[10] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, , and R. Morris. Event-driven programming for robust software. In *Proceedings of the SIGOPS European Workshop*. ACM, 2002.

[11] C. Elliott. Functional implementations of continuous modeled animation. In *International Symposium on Object Technologies for Advanced Software and Algebraic and Logic Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 284–299. Springer, 1998.

[12] C. Elliott. Declarative event-oriented programming. In *Proceedings of International Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 56–67. ACM, 2000.

[13] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of International Conference on Functional Programming (ICFP)*, volume 32 (8) of *SIGPLAN Notices*, pages 263–273. ACM, 1997.

[14] M. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *Proceedings of European Conference on Object-oriented Programming (ECOOP)*, volume 1445 of *Lecture Notes in Computer Science*, pages 186–211. Springer, 1998.

[15] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proceedings of Programming Language Design and Implementation (PLDI)*, volume 33 (5) of *SIGPLAN Notices*, pages 236–248. ACM, 1998.

[16] B. N. Freeman-Benson. Kaleidoscope: Mixing objects, constraints and imperative programming. In *Proceedings of Object-oriented Programming Systems, Languages, and Applications (OOPSLA) and European Conference on Object-oriented Programming (ECOOP)*, volume 25 (10) of *SIGPLAN Notices*, pages 77–88. ACM, 1990.

[17] B. N. Freeman-Benson and A. Borning. Integrating constraints with an object-oriented language. In *Proceedings of European Conference on Object-oriented Programming (ECOOP)*, volume 615 of *Lecture Notes in Computer Science*, pages 268–286. Springer, 1992.

[18] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *Proceedings of European Conference on Object-oriented Programming (ECOOP)*, volume 707 of *Lecture Notes in Computer Science*, pages 406–431, 1993.

[19] A. Goldberg and D. Robson. *SmallTalk-80: The Language and its Implementation*. Addison Wesley, Boston, MA, USA, 1983.

[20] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer, 2002.

[21] IBM. The Eclipse project. http://www.eclipse.org/.

[22] JGuru. The ANTLR parsing system. http://www.antlr.org/.

[23] G. Lopez, B. N. Freeman-Benson, and A. Borning. Implementing constraint imperative programming languages: The Kaleidospace'93 virtual machine. In *Proceedings of Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 29(10) of *SIGPLAN Notices*, pages 259–271. ACM, 1994.

[24] D. C. Luckham, J. Vera, and S. Meldal. Three concepts of system architecture. Technical report, Stanford University, 1995.

[25] J. Maloney. *An Introduction to Morphic: the Squeak User Interface Framework*, chapter 2, pages 39–68. Prentice Hall, Upper Saddle River, NJ, USA, 2002.

[26] J. H. Maloney and R. B. Smith. Directness and liveness in the Morphic user interface construction environment. In *ACM Symposium on User Interface Software and Technology*, pages 21–28. ACM, 1995.

[27] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proceedings of Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 37 (11) of *SIGPLAN Notices*, pages 211–222. ACM, 2001.

[28] N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of off-the-shelf components in C2-style architectures. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 692–700. IEEE Computer Society, 1997.

[29] T. Millstein. Practical predicate dispatch. In *Proceedings of Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 345–364. ACM, 2004.

[30] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, pages 51–64. ACM, Oct. 2002.

[31] D. Notkin, D. Garlan, W. G. Griswold, and K. J. Sullivan. Adding implicit invocation to languages: Three approaches. In *International Symposium on Object Technologies for Advanced Software*, volume 742 of *Lecture Notes in Computer Science*, pages 489–510. Springer, 1993.

[32] M. A. Orgun and W. Ma. An overview of temporal and modal logic programming. In *Proceedings of International Conference on Temporal Logic (ICTL)*, volume 827 of *Lecture Notes in Computer Science*, pages 445–479. Springer, 1994.

[33] J. Peterson, P. Hudak, and C. Elliott. Lambda in motion: Controlling robots with Haskell. In *Proceedings of Practical Applications of Declarative Languages (PADL)*, volume 1551 of *Lecture Notes in Computer Science*, pages 91–105. Springer, 1999.

[34] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *Proceedings of Operating System Design and Implementation (OSDI)*, pages 347–360. USENIX Association, 2000.

[35] M. Sage. FranTk - a declarative GUI language for Haskell. In *Proceedings of International Conference on Functional Programming (ICFP)*, volume 35(9) of *SIGPLAN Notices*, pages 106–117. ACM, 2000.

[36] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, Cambridge, MA, USA, 1986.

[37] Sun Microsystems, Inc. *The Java Collections API*. http://java.sun.com/j2se/1.5.0/-docs/guide/collections/.

[38] Sun Microsystems, Inc. *The JavaBeans Components API*. http://java.sun.com/-products/javabeans/.

[39] Sun Microsystems, Inc. *The JavaMail API*. http://java.sun.com/j2se/1.5.0/docs/-guide/collections/.

[40] Sun Microsystems, Inc. *The Swing API*. http://java.sun.com/products/jfc/.

[41] P. Tarau. Fluents: A refactoring of Prolog for uniform reflection an interoperation with external objects. In *Computational Logic*, pages 1225–1239, 2000.

[42] T. Teitelbaum and T. W. Reps. The Cornell program synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, 1981.

[43] T. A. Wagner and S. L. Graham. Efficient and flexible incremental parsing. *ACM Transactions on Programming, Languages, and Systems*, 20(5):980–1013, 1998.

[44] M. Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):74–84, 1993.