SuperGlue: Component Programming with Object-oriented Signals

Sean McDirmid[1] and Wilson C. Hsieh[2]

[1] École Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland
sean.mcdirmid@epfl.ch
[2] University of Utah, 84112 Salt Lake City, UT, USA
wilson@cs.utah.edu

Abstract. The assembly of components that can handle continuously changing data results in programs that are more interactive. Unfortunately, the code that glues together such components is often difficult to write because it is exposed to many complicated event-handling details. This paper introduces the SuperGlue language where components are assembled by connecting their **signals**, which declaratively represent state as time-varying values. To support the construction of interactive programs that require an unbounded number of signal connections, signals in SuperGlue are scaled with object-oriented abstractions. With SuperGlue's combination of signals and objects, programmers can build large interactive programs with substantially less glue code when compared to conventional approaches. For example, the SuperGlue implementation of an email client is around half the size of an equivalent Java implementation.

1 Introduction

Programs that are interactive are more usable than their batch program counterparts. For example, an interactive compiler like the one in Eclipse [15] can continuously detect syntax and semantic errors while programmers are typing, while a batch compiler can only detect errors when it is invoked. Interactive programs are often built out of components that can recompute their output state as their input state changes over time. For example, a compiler parser component could incrementally recompute its output parse tree according to changes made in its input token list. Other examples of these kinds of components include many kinds of user-interface widgets such as sliders and tables.

The assembly of components together in interactive programs often involves expressing state-viewing relationships. In object-oriented languages, such relationships are often expressed according to a model-view controller [13] (MVC) architecture. An MVC architecture involves model and view components, and *glue code* that transforms model state into view state. This glue code is often difficult to develop because most languages lack good constructs for transforming state. Instead, changes in state are often communicated as discrete events that must be manually translated by glue code into discrete changes of the transformed state.

This paper introduces SuperGlue, which simplifies component assembly by hiding event handling details from glue code. Components in SuperGlue are assembled by connecting together their *signals* [10], which represent state declaratively as time-varying

```
var folder : Mailbox.Folder;
if (folderView.selected.size = 1 &&
    folderView.selected[0] = folder)
  messageView.rows = folder.messages;
```

Fig. 1. SuperGlue code that implements email client behavior.

values. Operations on signals are also signals whose values automatically change when the values of their operand signals change. For example, if `x` and `y` are signals, then `x + y` is a signal whose current value is always the sum of the current values for `x` and `y`. By operating on signals to produce new signals, SuperGlue code can transform state between components without expressing custom event handlers.

Although signals are abstractions in functional-reactive programming languages [8, 10, 14], SuperGlue is novel in that it is the first language to combine signals with object-oriented abstractions. A program is expressed in SuperGlue as a set of signal connections between components. Because realistic programs often require an unbounded number of connections, each connection cannot be expressed individually. Instead, rules in SuperGlue can express new connections through type-based pattern matching over existing connections. To organize these rules, the types that are used in connection pattern matching are supported with three object-oriented mechanisms:

– **Nesting**, which is used to describe complicated components whose interfaces contain an unbounded number of signals. For example, inner node objects can describe the signals for an unbounded number of nodes in a user-interface tree component.
– **Traits**, which are used to integrate otherwise incompatible objects. For example, a trait can be used to describe how any kind of object is labeled in a user interface.
– **Extension**, which is used to implicitly prioritize connections to the same signal. For example, the connection of true to a bird object's `canFly` signal is of a lower priority than the connection of false to a penguin object's `canFly` signal.

These mechanisms form a novel object system that is specifically designed to support connection-based component programming.

As a concrete example of how SuperGlue can reduce the amount of code needed to assemble components together, consider the SuperGlue code in Figure 1, which implements the following master-detail behavior in an email client: "the messages of a folder that is uniquely selected in a folder view tree are displayed as rows in a message view table." This code glues together the **`folderView`** and **`messageView`** components, which are respectively a user-interface tree and table. The second line of Figure 1 is a condition that detects when only one node is selected in the folder view. The third line of Figure 1 is a *connection query* that detects if the first (and only) node selected in the folder view is connected to an email folder. If the connection query is true, the connected-to email folder is bound to the `folder` variable that is declared on the first line of Figure 1. If both of these conditions are true, the fourth line of Figure 1 connects the `rows` signal of the message view to the `messages` signal of the selected email folder.

Because the code in Figure 1 is evaluated continuously, how the rows of the message view table are connected can change during program execution. The user could select more than one node in the 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 with only four lines of SuperGlue code. In contrast, implementing this behavior in Java requires more than thirty lines of code because the code is exposed to event-handling details.

SuperGlue components are implemented with either SuperGlue code or Java code. When implemented with Java code, signals are represented with special Java interfaces that enable the wrapping of existing Java libraries. For example, we have implemented SuperGlue components around Java's Swing [24] and JavaMail [23] class libraries. The advantage of this dual language approach is that an expressive language (Java) can be used to implement components, while modularity is significantly enhanced by having components interact through SuperGlue's declarative signals.

This paper describes SuperGlue and how it reduces the amount of glue code needed to build interactive programs out of components. Section 2 details why components are difficult to assemble together in interactive programs. Section 3 introduces SuperGlue and provides examples of how it is used. Section 4 evaluates SuperGlue through a case study that compares an email client implementation in both SuperGlue and Java. Section 5 describes SuperGlue's syntax, semantics, and implementation. Section 6 presents related work and Section 7 summarizes our conclusions.

2 Motivation

We use the implementation of an email client program to show how components in interactive programs are difficult to assemble together. Consider the Java code in Figure 2, which assembles user-interface and email components to implement the following master-detail behavior: "the messages of an email folder that is uniquely selected in the folder view tree are the rows of a message view table." Two observers, which are object-oriented event handlers, are implemented and installed in Figure 2 to implement this behavior. The observer stored in the `messageObserver` variable translates folder message addition and removal events into table view row addition and removal events. The `messageObserver` is installed on an email folder object by the observer stored in the `selectionObserver` variable, which in turn is installed on the folder view tree (`folderView`). The `selectionObserver` is implemented to determine when the following condition is true or false: "only one node is selected and this node is an email folder." When this condition becomes true or false, the `messageObserver` is installed or uninstalled on the right email folder.

The Java code in Figure 2 is complicated because of its direct involvement in how state-change events are communicated between components. This involvement is necessary for two reasons. First, the way components transmit and receive events is often incompatible. In Figure 2, the email message addition and removal events transmitted from an email folder cannot be directly received by a user interface table. Second, events

```
messageObserver = new MessageCountListener() {
  void messageAdded(Message message)
  { messageViewModel.notifyRowInserted(message); }
  void messageRemoved(Message message)
  { messageViewModel.notifyRowDeleted (message); }
};
selectionObserver = new TreeSelectionListener() {
  Folder selected;
  void selectionAdded  (Object node) {
    int  selCount = folderView.getSelectedCount();
    Object selAdd = folderView.getSelected(0);
    if (selCount == 2 && selected != null) {
      selected.removeMessageCountListener(messageObserver);
      messageViewModel.notifyRowsChanged();
      selected = null;
    } else if (selCount == 1 && selAdd instanceof Folder) {
      selected = (Folder) selAdd;
      selected.addMessageCountListener(messageObserver);
      messageViewModel.notifyRowsChanged();
    }
  }
  void selectionRemoved(Object node) { ... }
};
folderView.addSelectionListener(selectionObserver);
```

Fig. 2. Java glue code that implements and installs the observerobjects of a message view component.

often affect state transformations in ways that must be translated manually. In Figure 2, tree node selection events are transformed into a condition that determines what email folder's messages are displayed. Detecting the discrete boundaries where this condition changes requires substantial logic in the `selectionObserver` implementation. For example, when a new node is selected in the tree view, code is needed to check if one folder was already selected, in which case the condition becomes false, or if one folder has become selected, in which case the condition becomes true.

Two approaches can currently be used to improve how the glue code of an interactive program is written. First, programming languages can reduce event handler verbosity with better syntax; e.g., through closure constructs and dynamic typing. Programming languages that follow this approach include Python [25], Ruby [19], and Scala [21]. However, although glue code in these languages can be less verbose than in Java, it is often not less complicated because programmers must still deal with the same amount of event handling details. In the second approach, standardized interface can be used in component interfaces to hide event handling details from glue code. For example, The *`ListModel`* Swing interface listed in Figure 3 can be used standardize how element addition and removal events are transmitted and received by different

```
interface ListModel {
  int getSize();
  Object getElementAt(int index);
  void    addDataListener(ListDataListener listener);
  void removeDataListener(ListDataListener listener);
}
```

Fig. 3. Swing's `ListModel` Java interface, which describes a list with observable changes in membership.

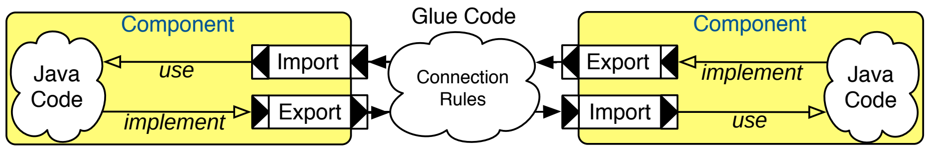


Fig. 4. An illustration of a SuperGlue program's run-time architecture; "use" means the Java code is using imported signals through a special Java interface; "implement" means Java code is providing exported signals by implementing special Java interface.

components. Using list model interfaces in our email client example, displaying email messages as rows in a table view can be reduced to the following line of code:

```
tableView.setRows(folder.getMessages());
```

Unfortunately, standardized interfaces cannot easily improve glue code that performs state transformations. For example, the glue code in Figure 2 selects what folder's messages are displayed using a condition. Expressing this condition with standardized interfaces requires redefining how `==`, `&&`, and, most significantly, **`if`** operations work. Although the resulting code could be syntactically expressed in Java, such code would be very verbose and not behave like normal Java code.

3 SuperGlue

The problems that are described in Section 2 occur when glue code is exposed to events that communicates state changes between components. When event handling can be hidden from glue code with standardized interfaces, these problems disappear and glue code is much easier to write. However, the use of standardized interfaces to express state transformations in glue code requires layering a non-standard semantics on top of the original language. Instead, these standardized interfaces should be supported with their own syntax. In SuperGlue, standardized interfaces are replaced with **signal** language abstractions that represent mutable state declaratively as time-varying values.

The architecture of a SuperGlue program is illustrated in Figure 4. A SuperGlue program is assembled out of components that interact by viewing each others' state through signals. A component views state through its imported signals, and provides

```
atom Thermometer {
  export temperature : Int;
}
atom Label {
  import text  : String;
  import color : Color;
}
```

Fig. 5. Declarations of the `Thermometer` and `Label` atoms.

state for viewing through its exported signals. SuperGlue code defines program behavior by operating on and connecting the signals of the program's components together. Components in SuperGlue can be implemented either in SuperGlue or Java code, while components are always assembled together with SuperGlue code. Components are implemented in Java code according to special Java interfaces that are described in Section 5. For the rest of this section, we focus on the SuperGlue code that assembles components together.

A component in SuperGlue is an instance of either an *atom*, which is implemented with Java code, or a *compound*, which is implemented with SuperGlue code. Two example atoms are declared in Figure 5. Signals are declared in a component to be either exported or imported and are associated with a type. The `Thermometer` atom declares an exported `temperature` signal that is the value that a thermometer component measures. The `Label` atom declares an imported `text` signal that is the value that is displayed by a label component. The `Label` atom also declares an imported `color` signal that is the foreground color of a label component.

Components in a program are instantiated atoms or compounds. For example, the following code instantiates the `Thermometer` atom to create the `model` component:

```
let model = new Thermometer;
```

Interactions between components are established by connecting their signals together. Signal connection syntax in SuperGlue resembles assignments in a C-like language: the left-hand side of a signal connection is an imported signal that is connected to the right-hand side of a signal connection, which is an expression. As an example of a connection, consider the following glue code:

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

This code connects the `text` signal that is imported into the `view` component to an expression that refers to the `temperature` signal that is exported from the `model` component. Because of this connection and the Java implementations of the `Label` and `Thermometer` atoms, whenever the temperature measured by the `model` component changes, the text displayed in the `view` component is updated automatically to reflect this change.

Connections are expressed in rules with conditions that guard when the connections are able to connect signals. When all the conditions of a rule evaluate to true, the rule is

```
if       (model.temperature > 30) view.color = red;
else if  (model.temperature < 0 ) view.color = blue;
else                              view.color = black;
```

Fig. 6. Glue code that causes the color of the `view` label component to change according to the current temperature measured through the `model` thermometer component.

active, meaning that the source expression of its signal connection can be evaluated and used as the sink signal's value. Connection rules in SuperGlue are expressed as C-like `if` statements. As an example of a rule, consider the following code:

```
if (model.temperature > 30) view.color = red;
```

This code connects the foreground color of the `view` component to the color red when the current temperature measured by the `model` component is greater than `30`. When the current temperature is not greater than `30`, the condition in this code prevents red from being used as the for the foreground color of the `view` component.

Although rules in SuperGlue resemble `if` statements in an imperative language, they have semantics that are declarative. Conditions are evaluated continuously to determine if the connections they guard are active. In our example, the current temperature can dynamically go from below `30` to above `30`, which causes the `view` component's foreground color to become red. In SuperGlue's runtime, this continuous evaluation is transparently implemented with event handling that activates the port connection when the current temperature rises above `30`.

Multiple rules that connect the same signal form a *circuit* that controls how the signal is connected during program execution. At any given time, any number of rules in a circuit can be active. If all rules in a signal's circuit are inactive, then the signal is unconnected. If exactly one rule in a circuit is active, then the circuit's signal is connected according to that rule. It is also possible that multiple rules in a circuit are active at the same time, while only one of these rules can connect the circuit's imported signal. If these rules cannot be prioritized, then the signal is connected ambiguously.

To explicitly prioritize rules in a circuit, rules can be expressed in the body of an `else` clause so that they are never active when rules that are expressed in the body of the corresponding `if` clause are active. As an example of a circuit, consider the glue code in Figure 6, which continuously connects a different color to the `view` label's foreground color depending on the current temperature. As the current temperature falls below `30`, the foreground color of the `view` label changes from red to black. Likewise, as the current temperature falls below `0`, the foreground color of the `view` label changes from black to blue. The color connection code in Figure 6 forms a circuit that is illustrated in Figure 7. Conditions, which are ovals, control whether a connection is active based on their test inputs. Multiple connections are organized into a switch that passes through the highest priority connection, which is closest to `hi`, that is currently active.

With the basic connection model that has been described in this section, each signal connection in a program is encoded separately. This model has two limitations:

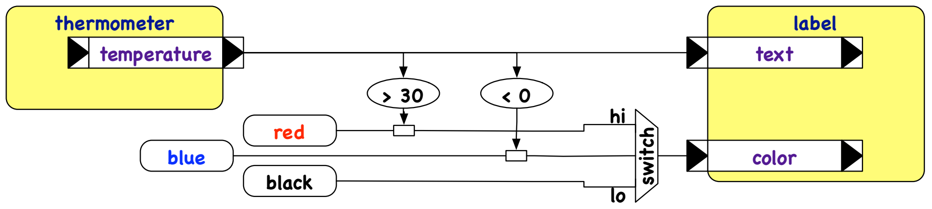


Fig. 7. An illustration of a circuit that connects to the `color` signal of the **`view`** label component; rounded rectangles are components; boxes that end with triangles are signals; ovals are conditions with outgoing results; and small boxes activate connections according to their incoming condition results; the multiplexor “switch” only passes through the highest (closest to `hi`) active connection.

– Programs that deal with stateful graph-like structures such as lists and trees cannot be expressed very effectively. Graph-like structures are unbounded in their sizes and therefore cannot be expressed as a fixed number of direct connections between components.
– 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.

If unaddressed, these two limitations prevent the effective construction of real programs in SuperGlue. Consider the email client that is used as an example in Section 2. This email client consists of a user-interface tree that displays a hierarchy of mailboxes and email folders. Because the size of this tree is not known until run-time, it cannot be expressed with a fixed number of connections. Additionally, the connections in this tree are largely homogeneous; e.g., the connections used to relate one tree node to one email folder are repeated to relate another tree node to another email folder. As a result, specifying each connection individually would result in very repetitive code.

3.1 Connection Nesting and Reification

Connections are not as expressive procedures, and so by themselves cannot scale to express non-trivial programs. In a way that is similar to how procedures are enhanced by being organized into objects, we enhance connections in SuperGlue with object-oriented abstractions to improve their expressiveness. In SuperGlue, connections can be identified at run-time by the types of the signals they connect. A rule can then create a new connection relative to any existing connection that matches a specified type pattern. SuperGlue supports such type-based pattern matching with object-oriented abstractions. Objects in SuperGlue serve two roles: first, they are containers of imported and exported signals; and second, they serve as nodes in a program’s connection graphs. Each signal connection in SuperGlue involves objects that reify the import being connected and the expression that the import is being connected to. The types of these objects are then used to identify the connection when rules are evaluated. To better support the use of

```
atom TreeView {
  inner Node {
    import text : String;
    import children : List[T : Node];
  }
  import root     : Node;
  export selected : List[T : Node];
}
```

Fig. 8. The TreeView atom and the Node inner object type that is nested in the TreeView atom.

```
atom Mailbox {
  inner Message {...}
  inner Folder {
    export sub_folders : List[T : Folder];
    export messages     : List[T : Message];
  }
  export root_folder   : Folder;
}
```

Fig. 9. The Mailbox atom and the Message and Folder inner object types that are nested in the Mailbox atom.

objects as types in connections, object are supported with nesting, which is described next, traits, which is described in Section 3.2, and extension, which is described in Section 3.3.

Object nesting is similar to pattern nesting in BETA [17] and is used to describe components that contain a large or unbounded number of signals. As an example, the TreeView atom is declared in Figure 8 with the Node inner object type, which is used to represent user-interface tree nodes. A TreeView component imports a root Node object, and each of its Node object imports its own list of child Node objects. In this way, inner Node objects can be used to concisely describe the unbounded hierarchical structure of a user-interface tree. The *List* trait used in Figure 8 describes lists of objects of a type specified in brackets. A signal of type *List*[T : Node] contains an integer size signal and a signal of type Node for each of its elements; e.g., the children signal contains [0] and [1] signals, which are both of type Node. We describe the *List* trait in Section 3.3.

At run-time, an object in SuperGlue is a vertex in the program's connection graph that is connected to other objects. The declared type of a signal describes the object that is attached to the signal. For example, an imported root signal of declared type Node is attached to a Node object. The type of a signal does not restrict how the signal can be used in a connection. In fact, the types involved in the same connection do not have to be related in any way. Instead, types describe how objects are connected as a result of connections. For example, consider the following SuperGlue code, which uses the Mailbox atom that is declared in Figure 9:

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

This rule connects an exported `root_folder` signal of type `Folder` to an imported `root` signal of type `Node`. The `Folder` and `Node` types of the connection's objects are entirely unrelated. Despite the unrelated types, this connection is allowed because other rules can resolve the incompatibilities that occur between these two objects.

Unlike components, inner objects are not instantiated in glue code. Instead, the creation and identity of an inner object is managed inside its containing component. As a result, inner object types can describe a component's interface without revealing details about how the component is implemented. For example, the folder and message objects nested of a mailbox components can be implemented to only have allocated objects while they are being used.

SuperGlue code can abstract over objects based on their types using SuperGlue's variable construct. As an example, consider the following code:

```
var node : folderView.Node;
```

This code declares a `node` variable, which abstracts over all `Node` objects in the **`folderView`** component. Variables must be bound to values when they are used in rules. A variable is bound to a value when it is used as a connection *target*, which means its signal is being connected in the connection rule. As an example of how a variable is bound when it is a connection target, consider the following code:

```
var node   : folderView.Node;
node.children = folder.sub_folders;
```

Because the `node` variable in this code is the target of a connection, it is always bound to some tree node object when the rule is evaluated. The way that variables are bound when they are used as connection targets resembles how procedure arguments are bound in a procedural language, where evaluating a connection binds the connection's target to a value is analogous to how calling a procedure binds the procedure's arguments to values.

In conjunction with variables, rules can abstract over connections in a program by identifying how objects are connected with *connection queries*. A connection query is a condition of the form `sink = source`, where `sink` and `source` are expressions. A connection query succeeds only if `sink` is connected to a value that is equivalent to `source`, where unbound variables referred to in `source` can become bound to facilitate this equivalence. Whether a variable can be bound to a value depends on the value's type being compatible with the variable's type. As an example, consider the following code:

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

This code is evaluated as follows:

```
atom TableView {
  inner Row { ... }
  import rows : List[T : Row];
}
let messageView = new TableView;
var folder : Mailbox.Folder;
if (folderView.selected.size = 1 &&
    folderView.selected[0] = folder)
  messageView.rows = folder.messages;
```

Fig. 10. The `TableView` atom and code that describes how the rows of a message view table are connected in an email client.

1. The `node` variable, which is the target of the rule defined in this code, is bound to a `Node` object whose `children` signal is being accessed.
2. The connection query `node = folder` checks if the targeted `Node` object is connected to a `Folder` object of a `Mailbox` component. If it is, then the `folder` variable is bound to the connecting `Folder` object.
3. If the connection query is true, the imported `children` signal of the `Node` object is connected to the exported `sub_folders` signal of the `Folder` object.

Variables and connection queries allow rules in SuperGlue to be reused in multiple connection contexts. For example, because of the following rule, the root node object of the folder view tree is connected to the root folder object of a mailbox:

```
folderView.root = mailbox.root_folder;
```

When the folder view tree's implementation accesses the `children` signal of its root node object, the `node` variable target of the following rule, which was just described, is bound to the root node object:

```
if (node = folder) node.children = folder.sub_folders;
```

Because of the connection query in this rule, the `folder` variable is bound to the root folder object of the mailbox object. As a result, the `children` signal of the root node object is connected to the root folder object's `sub_folders` signal. In turn, the same rule connects the `folderView.root.children[0].children` signal to the `mailbox.root_folder.sub_folders[0].sub_folders` signal, and so on. Despite the recursion, this rule will terminate because it is only evaluated when a tree node is expanded by a user.

Our use of variables and connection queries to build trees resembles how hierarchical structures are often built in user interfaces using conventional object-oriented languages. For example, in Java's Swing [24] library, `TreeModel` objects often examine the run-time types of objects, which are used as nodes, to compute their child nodes. Compared to this approach, expressing a tree in SuperGlue requires less code because of SuperGlue's direct support for expressing connection graphs.

SuperGlue's signal and object-oriented abstractions are designed to work seamlessly together. Circuits that change how signals are connected at run-time consequently change how objects are connected. Additionally, because signal expressions can be used as the source of a connection query, if and how a connection query is true can change at run-time. As an example, consider the SuperGlue code in Figure 10, some of which was presented in Section 1. The second condition of the rule defined in this code is a connection query that tests if the first and only selected node of a folder view tree is connected to a folder object. As node selection in the folder view changes, the condition can become false if the newly selected node is not a folder object. Alternatively, a different folder object can become bound to the `folder` variable if the newly selected node is another folder. The latter situation would cause the rows of the message view table to be connected to a different list of email messages, which in turn causes the message view table to be refreshed.

3.2 Traits

SuperGlue traits are similar to Java interfaces in that they lack implementations. Traits serve two purposes: they enable the reuse of signal declarations in different component prototypes; and they enable *coercions* that adapt incompatible objects by augmenting them with new behavior.

Unlike the signals that are declared in atoms and compounds, signals are declared in traits with the `port` keyword, meaning whether they are imported or exported is not fixed. As an example, the following SuperGlue code declares the `Labeled` trait with one `label` signal:

```
trait Labeled { port label : String; }
```

An atom or compound can extend a trait by specifying if the trait's signals are imported or exported. As an example, the following code declares that the `Node` inner object type imports the signals of the `Labeled` trait:

```
atom TreeView { inner Node imports Labeled { ... } }
```

Whenever an object that imports a trait is connected to another object that exports the same trait, the trait's declared signals in both objects are connected by default. As an example, first consider having the `Folder` inner object type of a **`Mailbox`** atom export the `Labeled` trait:

```
atom Mailbox { inner Folder exports Labeled { ... } }
```

When `Node` objects are connected to `Folder` objects, the rule `node.label = folder.label` is implied and does not need to be specified in glue code.

The example in the previous paragraph is not very modular: the `Folder` inner object type, which is an email concern, should not implement the `Labeled` trait, which is a user-interface concern. In general, components should only declare signals that are necessary to express their state and capabilities. Traits for other concerns can be externally implemented as coercions by using variables and connection queries (Section 3.1)

```
atom Mailbox {
  inner Folder { export name : String; }
  ...
}
var labeled : Labeled;
var folder  : Mailbox.Folder;
if (labeled = folder) labeled.label = "Folder " + folder.name;
```

Fig. 11. An example of how a `Labeled` trait coercion is defined for `Folder` objects that are nested in `Mailbox` components.

to specify how the components' signals are translated between the traits' signals. As an example of a coercion, consider the SuperGlue code in Figure 11, where the `Folder` object does not implement the `Labeled` trait. Instead, the rule in Figure 11 specifies how the `label` signal of an object that imports the `Labeled` trait is connected when the object is connected to a `Folder` object. This rule then applies to each tree `Node` object that is connected to an email `Folder` object.

3.3 Extension

SuperGlue's support for type extension enables the refinement of inner object types and traits. Extension in SuperGlue differs from extension in conventional object-oriented languages in the way that it is used to prioritize rules in circuits. Such prioritization is the only way in SuperGlue to prioritize rules that are specified independently.

Given two SuperGlue variables, one variable is more specific than the other variable if the type of the former variable extends the type of the latter variable. Connections are then prioritized based on the specificity of their involved variables. As an example, consider the following code:

```
trait Bird { port canFly : Boolean; }
trait Penguin extends Bird;
var aBird : Bird;
var aPenguin : Penguin;
aBird.canFly = true;
aPenguin.canFly = false;
```

The `Penguin` trait extends the `Bird` trait, so the type of the `aPenguin` variable is more specific than the type of the `aBird` variable. As a result, the connection of a penguin object's `canFly` signal to `false` has a higher priority than the connection of a bird object's `canFly` signal to `true`. In this way, penguin behavior **overrides** more generic bird behavior.

Besides being used to prioritize connections, type extension is also used in SuperGlue to refine inner object types. Generic typing is achieved in SuperGlue by refining inner object types when their containers are extended. As an example, the `List` trait, which is declared in Figure 12, describes list items through its `T` inner object type. The `T` inner object type can be refined whenever the `List` trait is extended, imported, or exported. As an example, consider the following code:

```
trait List extends Array {
  inner T;
  port size : Int;
  port index : Int;
  port item  : T;
}
```

Fig. 12. The declaration of the **List** trait.

```
atom TableView {
  inner Row { ... }
  inner Rows imports List {
    refine T extends Row;
  }
  import rows : Rows;
}
```

In this code, the `Rows` inner object type is declared to extend the *`List`* trait through the **`imports`** keyword. As a result, the `Rows` inner object type has the *`List`* trait's `T` inner object type, which itself can be refined to extend the `Row` inner object type declared in the **`TableView`** component prototype. As a result, any element of the rows list will be an object that is or is connected from a row object. The refinement of a single inner object type in a trait is a common operation and is supported by SuperGlue with syntactic sugar so that the above SuperGlue code can be re-written as follows:

```
atom TableView {
  inner Row { ... }
  import rows : List[T : Row];
}
```

In this code, the colon operator is used twice as a short hand for extension: the colon expresses that the `T` inner object type from the *`List`* trait extends `Row` and that the type of the `rows` signal imports the *`List`* trait. Similar syntax was used to declare list signals in Figure 8, Figure 9, and Figure 10.

3.4 Array Signals and Streams

Multi-element data is expressed in SuperGlue with signals whose types extend the built-in *`Array`* trait, which we refer to as *array signals*. The *`List`* trait is declared in Figure 12 to extend *`Array`* so any signal that is declared with the *`List`* trait will be an array signal. Array signals are similar to SQL tables–an individual element of an array can only be accessed through an SQL-like query that selects the element according to the values of its sub-signals. As an example, the following SuperGlue code selects the first element of a table's list of rows:

```
table.rows(index = 0).item
```

This SuperGlue code is similar in behavior to the following pseudo-SQL query:

```
SELECT item FROM table.rows WHERE index = 0
```

Alternatively, SuperGlue provides syntactic sugar to access list elements; e.g., `table-.rows[0]` is equivalent to the above SuperGlue code. Array signal queries are also used to express arithmetic expressions. For example, the expression `x + y` is syntactic sugar for `x.plus(operand = y).result`. We use query syntax rather than argument binding for an important reason: the coercions that are described in Section 3.2 can be directly applied to query bindings, which would be problematic with parameter passing semantics.

So that entire interactive programs can be expressed in SuperGlue, SuperGlue supports *streams* that can be used to manipulate discrete events and imperative commands. There are two kinds of streams in SuperGlue: event streams, which intercept discrete events, and command streams, which perform imperative commands. As an example of how streams are used, consider the following SuperGlue code that implements delete email message behavior:

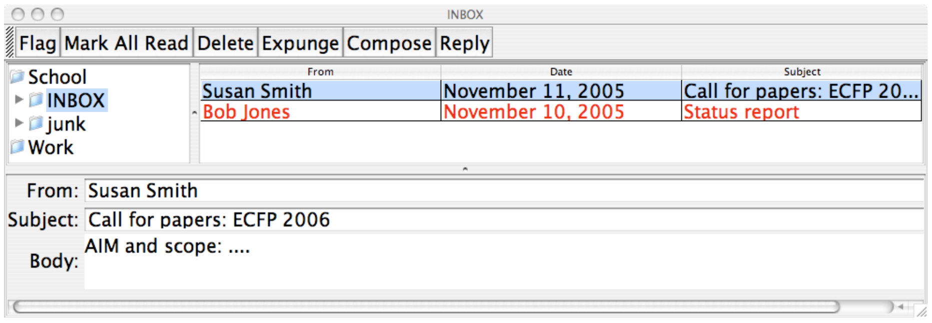
```
let delete_button = new Button;
on (delete_button.pushed)
  if (msg : Mailbox.Message = messageView.selected)
    do msg.delete;
```

The last three lines of this code use streams. The **on** statement intercepts events where the delete button is pushed. When the delete button is pushed, each email message selected in the message view table is deleted by executing its `delete` command stream in a **do** statement. Streams in SuperGlue are convenience abstractions that enable imperative programming without sacrificing the benefits of SuperGlue's signals. Unlike signals, we do not claim that there is a significant advantage to using SuperGlue to express components interactions through streams.

4 Evaluation

Our evaluation of SuperGlue focuses on our claim that SuperGlue can substantially reduce the amount of glue code needed to build an interactive program out of components. More importantly, we claim that this code reduction corresponds to a similar reduction in complexity. This section evaluates our claims using the case study of implementing a complete email client in SuperGlue, which has been used as our primary example in this paper. We then compare this SuperGlue implementation of an email client to a feature-equivalent implementation in Java. As part of this comparison, we have designed a strategy to compare complexity that accounts for differences between SuperGlue and Java syntax.

Our email client is composed of user-interface and email components. For this case study, we have implemented the necessary SuperGlue component libraries by wrapping the Swing [24] and JavaMail [23] class libraries, which are used directly in the Java email client. How a Java class library is wrapped into a SuperGlue component library is



From	Date	Subject
Susan Smith	November 11, 2005	Call for papers: ECFP 20...
Bob Jones	November 10, 2005	Status report

From: Susan Smith
Subject: Call for papers: ECFP 2006
Body: AIM and scope: ....

Fig. 13. A screen shot of an email client that is implemented in SuperGlue.

described in Section 5. By far, the most complicated wrapping involves Swing's `JTree` and `JTable` classes, which require 444 and 611 lines of Java code, respectively. This code is devoted to translating the Java-centric abstractions of the original classes into SuperGlue signal abstractions. Component wrappers involve a lot of code that is not counted in our case study because the resulting component libraries are reusable in multiple applications. On the other hand, the need for wrapper code represents a significant amount of complexity that must be amortized by reusing the components in many programs.

A screen shot of the SuperGlue email client is shown in Figure 13. Our comparison case study is organized according to the code that 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 body view form, which views the contents of a message. In Figure 13, 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.
- **Composition**, which allows a user to compose and send a new message, and reply to an existing message.

The methodology used in our comparison case study involves measuring two metrics in each implementation: lines of code and number of operations. While line counts are accurate measures of verbosity, they are not necessarily accurate measures of complexity. Verbosity and complexity are only loosely related, and 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

	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

Fig. 14. A comparison of email client features as they are implemented in SuperGlue and Java.

example, a method call in Java or a signal connection in SuperGlue are both counted as an operation each, while variable uses in both Java and SuperGlue are not counted as operations. Because the operations we count are related to using a library, they are a more accurate measure of complexity than line count.

The results of our comparison are shown in Figure 14. In these results, the line count and operation metrics are similar for each feature so, at least in this case study, operation density per line is similar between SuperGlue and Java. By far the largest reduction in program complexity is obtained in the SuperGlue implementation of the navigation feature. This reduction is large because the navigation feature uses trees, tables, and forms, which are components with a large amount of signal-based functionality. Besides the navigation feature, the deletion and composition features involve only a minor amount of continuous behavior, and so their implementations do not benefit very much from SuperGlue. Overall, SuperGlue's reduces the amount of code needed to implement an email client by almost half because the navigation feature requires much more Java code to implement than the other two features combined.

According to the results of this initial case study, SuperGlue can provide a code reduction benefit when an interactive program contains a significant amount of continuous behavior that can be expressed as signals. As mentioned in Section 3.4, SuperGlue's use is of little benefit in programs that involve a lot of non-continuous behavior; i.e., behavior that is discrete or imperative.

5 Syntax, Semantics, and Implementation

Our prototype of SuperGlue can execute many realistic programs, including the email client described in Section 4. Our prototype consists of an interpreter that evaluates the circuits of signals at run-time to drive component communication. This prototype can deal with components that are implemented either in SuperGlue (compounds) or Java (atoms). The discussion in this section describes informally the syntax, semantics, and implementation of SuperGlue. For space reasons, our discussion omits the following language features: **`else`** clauses, array signals, and streams. Our discussion also does not cover syntactic sugar that was described in Section 3.

5.1 Component and Trait Declarations

The syntax used to declare components and traits is expressed informally in Figure 15. **Notation:** the horizontal over-arrow indicates zero-to-many repetition, while the vertical bar indicates choice. A program consists of a collection of atoms, compounds, and

$$
\begin{array}{lll}
\mathit{program} & \equiv & \overrightarrow{\mathit{atom} \mid \mathit{compound} \mid \mathit{trait}} \\
\mathit{atom} & \equiv & \mathbf{atom}\ \mathbf{AtomName}\ \mathit{decls} \\
\mathit{compound} & \equiv & \mathbf{compound}\ \mathbf{CompoundName}\ \mathit{decls}\ \mathbf{with}\ \{\ \mathit{glue\text{-}code}\ \} \\
\mathit{trait} & \equiv & \mathbf{trait}\ \mathit{TraitName}\ \mathit{decls} \\
\mathit{decls} & \equiv & \overrightarrow{\mathit{implements}}\ \{\ \overrightarrow{\mathit{inner} \mid \mathit{signal}}\ \} \\
\mathit{inner} & \equiv & (\mathbf{inner} \mid \mathbf{refine})\ \mathtt{InnerName}\ \overrightarrow{\mathbf{extends}\ \mathtt{AnInner}}\ \mathit{decls} \\
\mathit{implements} & \equiv & (\mathbf{imports} \mid \mathbf{exports} \mid \mathbf{extends})\ \mathit{ATrait} \\
\mathit{signal} & \equiv & \\
\multicolumn{3}{l}{\quad (\mathbf{import} \mid \mathbf{export} \mid \mathbf{port})\ \mathtt{signalName}\ :\ (\mathit{ATrait} \mid \mathit{AnInner})}
\end{array}
$$

Fig. 15. Informal syntax for SuperGlue programs as well as components and traits.

traits. Atom s have Java implementations, which are described in Section 5.3. Compounds are implemented in SuperGlue code (`glue-code`) to glue other constituent components together. SuperGlue code is described in Section 5.2. SuperGlue code always exists in compound components, and a self-contained SuperGlue program is a compound that lacks imports and exports.

Atoms, compounds, traits, and inner object types all have declarations (`decls`) of their extended traits, signals, and inner object types. Sub-typing ($<_t$) is established by trait extension for components, traits, and inner object types, and by inner object type extension for inner object types. Inheritance behaves in the usual way; i.e., a type inherits all signals and inner object types of its super-types. An inner object type is a subtype of any inner object type it mirrors in its container's super-types: if **B** $<_t$ **A**, then **B**.*anInner* $<_t$ **A**.*anInner* if *anInner* is declared in **A**. Additionally, inner object types are extended virtually: if **A**.*anInnerY* $<_t$ **A**.*anInnerX* and **B** $<_t$ **A**, then **B**.*anInnerY* $<_t$ **B**.*anInnerX*. Because of this virtual extension behavior, as inner object types are refined via the **refine** keyword, all extending inner object types with the same container type automatically inherit the refinements.

When declaring a signal or extending a trait within the declaration of an atom or compound, the import/export polarity of the declared signal or the extended trait's signals must be specified via the **import(s)** and **export(s)** keywords. When declaring a signal or extending a trait within a trait declaration, import/export polarity is not specified so the **port** or **extends** keywords are used instead. When a trait is imported or exported into a component or inner object type of a component, the trait's signals are inherited under the same import/export polarity, which also applies to the signals of the trait's inner object types. Likewise, when a trait is used as a signal's type, the signal inherits the trait's signals with its own import/export polarity.

5.2 SuperGlue Code

SuperGlue code, which forms the body of a compound component, is a collection of let statements, variable declarations, and rules whose syntax are described in Figure 16. For brevity, we assume that each rule is expressed in its own single **if** statement, although **if** nesting is possible in the actual language. We also do not address **else** statements, which act to negate the conditions in a corresponding **if** clause.

```
glue-code ≡ $\overrightarrow{let | var | rule}$
rule      ≡ if ($\overrightarrow{query}$) $e_x$.aSignal = $e_y$;
let       ≡ let instanceName = new aComponent;
var       ≡ var varName : (aTrait | aComponent | e.anInner)
query     ≡ $e_v$ = $e_w$
e ≡ anInstance | aVar | e.aSignal | aConstant | anException
```

Fig. 16. Informal syntax for the SuperGlue code that is inside a compound component.

Expressions (e) in SuperGlue are values that have run-time representations. Evaluation determines how an expression is connected to another expression. Evaluation can be applied successively to determine what value that an expression is terminally connected to. This value will either be a constant as a number, string, handle, or an exception, which indicates failure in the evaluation.

The evaluation of a signal expression (e_z.`aSignal`) involves evaluating the signal's circuit of prioritized rules if the signal connected with SuperGlue code. The rules in a signal's circuit are prioritized according to the type specificity of each rule's target expression (e_x). Given two rules with target expressions e_a and e_b, the first rule is of a higher priority than the second rule if e_a's type is a subtype of e_b's type. If the target expressions of two rules do not have a sub typing relationship, then the rules have the same priority and are ambiguous.

To evaluate a rule, the e_x.`aSignal` expression of the rule is connected if possible to the evaluated signal expression e_z.`aSignal`. This is represented and evaluated as the connection query e_x `=` e_z, which is the implied first antecedent of every rule. Given a specific evaluated signal expression (e_z.`aSignal`), a rule is instantiated into the following:

```
if ($e_z$ = $e_x$ && $\overrightarrow{query}$) $e_z$.aSignal = $e_y$
```

The connection queries that are a rule's antecedents perform a uni-directional form of unification in logic programming languages: the query e_v `=` e_w is true if e_v is connected to an expression that is equivalent to e_w, where variables referred to by e_w can be bound to expressions in e_v to meet the equivalence condition. A connection query successively evaluates e_v until an expression is yielded that is equivalent to e_w, or until an end-point expression is reached, in which case the connection query fails.

A variable can be bound to an expression in a connection query if it is not already bound and the expression is of a type that is compatible with the variable's declared type. The binding of a variable does not cause the variable to be replaced with its bound-to expression. Instead, the variable (`aVar`) becomes connected to the bound-to expression (e) so that `aVar` evaluates to e.

A rule is active if all of its antecedents evaluate to true. If the highest-priority active rule of a circuit is unambiguous, then e_z.`aSignal` evaluates to e_a where e_y evaluates to e_a in an environment of any variable bindings that are created when the rule's connection queries are evaluated. If no rule is active, then the expression is unconnected. If multiple highest-priority rules are active, then the expression is connected to an am-

```
interface Signal {
  Value eval(Value target, Context context, Observer o);
  Signal signal(Value target, Port port);
}
interface Observer {
  void notify(Context context);
}
```

Fig. 17. Java interfaces used to implement signals.

biguity exception. Any other kind of exception that occurs during evaluation, such as dividing by zero, will also be propagated as the result of the evaluation.

If the evaluated signal expression e_z.`aSignal` lacks a circuit, which occurs when e_z is a variable, or its circuit evaluates to an unconnected exception, then e_z is evaluated into e_n. If e_n has a type that declares `aSignal`, then the expression e_n.`aSignal` is used as e_z.`aSignal`'s evaluation. In this way, we uniformally deal with variables and expressions, as well as establish default connections between container expressions that extend common traits, as described in Section 3.2.

5.3 Component Implementations

Signals that are imported into atoms or compounds or are exported from compounds are implemented with SuperGlue code. Evaluating a compound's exported signals is similar to evaluating imported signals, although the evaluation environments between the outside and inside of a compound are different. Signals that are exported from atoms are implemented with Java code. Atoms implement their exported signals and use their imported signals according to a Java interface that is described in Figure 17. This interface declares an `eval` method that allows an atom's Java code to implement the circuits of its exported signals directly and also enables access to the SuperGlue-based circuits of an atom's imported signals.

The `eval` method of the *`Signal`* interface accepts an observer that is notified when the result of the `eval` method has changed. When implementing a signal, Java code can install this observer on the Java resource being wrapped by the atom. For example, when implementing the exported selected nodes signal of a user-interface tree, this observer can be translated into a selection observer that is installed on a wrapped **`JTree`** object. When a signal is being used, Java code can provide an observer to notify the Java resource that is being wrapped of some change. For example, when using the imported rows signal of a user-interface table, an observer is implemented that translates imported row changes into `notifyRowsAdded` and `notifyRowsRemoved` calls on a wrapped **`JTable`** object.

An atom's Java code can either return arbitrary values when implementing an `eval` method or pass arbitrary values as targets when using an `eval` method. As a result, an atom has a lot of flexibility when manipulating inner objects. For example, an atom can create new inner objects on the fly and pass them into the right signal eval methods. This occurs when wrapping a `JTree` object, where parameters of methods in the tree model object are used as targets when querying the imported signals of inner `Node` objects.

5.4 Continuous Evaluation

The observers that are implemented in and used by Java code form the underlying infrastructure for *continuous evaluation* in SuperGlue, where evaluation results are updated continuously as state changes in the program's atoms. When an imported signal is evaluated from within an evaluating atom, the atom can specify an observer that is installed on dependent evaluations. Eventually, when evaluation reaches the exported signals of other atoms, this observer is installed in those atoms. When state changes in any of these other atom implementations, the evaluating atom is notified through the observer it defined. The evaluating atom can then re-evaluate the changed imported signal to refresh its view of state in other components.

When the state of an atom's imported signal changes, the atom will uninstall its observer on the signal's old evaluation and install its observer on the signal's new evaluation. The atom may also compute changes in the state of its own exports, where observers that are defined in other atoms and are installed on the atom are notified of these changes. A naive implementation of continuous evaluation processes all state changes as soon as they occur. However, this strategy results in *glitches* that cause atoms to observe inconsistent old and new signal evaluations. In the case of a glitch, an observer is not uninstalled and installed in the correct evaluation contexts, and therefore atoms will begin to miss changes in state.

Avoiding glitches in SuperGlue involves a sophisticated interpreter that adheres to the following guidelines:

– During the processing of a change in the evaluation of an atom's exported signal, ensure that both the signal's old and new evaluations are computable. This allows clients to compute old evaluations for derived signals so that observers can be uninstalled as necessary.
– The evaluation of an atom's exported signal cannot exhibit a new value until the corresponding change is processed and observers are notified of this new value. This is an issue when a signal changes rapidly so that the processing of its new evaluations could overlap.
– A change in the value of an atom's exported signal is processed in two phases. The first phase only discovers what exported signals in other atoms have changed as a result of the first exported signal's change. As a result of this discovery, the second phase updates observers for each changing signal together. The separation of these two phases allow multiple dependent signals to change together and avoid situations where they exhibit combinations of evaluations that are inconsistent; e.g., where both `b` and `!b` evaluate to true.

5.5 Cycles, Termination, and Performance

Cycles in the signal graph occur when the evaluation of a signal expression yields itself. In SuperGlue, cycles are detected at run-time and rejected with a cyclic connection exception. Although cycles created solely in SuperGlue code can be statically detected, a cycle can also occur because of dependencies between an atom's imported and exported signals. For example, a table view imports a list of rows and exports a list of these rows that are selected. Connecting the exported selected rows of a table view to its imported rows creates a cycle.

Even without cycles in the signal graph, non-termination can still occur in a SuperGlue program. Because SuperGlue lacks recursion that is strong enough to traverse data structures, SuperGlue code by itself will never be the sole cause of non-termination. However, atoms can be implemented with arbitrary Java code, and how the atom is connected by SuperGlue code can influence whether this Java code terminates. For example, a tree view atom will not terminate if it is configured to expand all tree nodes and is connected to a tree model of an unbounded size.

Our prototype implementation of SuperGlue is not tuned in any way for performance. According to microbenchmarks, SuperGlue code is between two and 144 times slower than equivalent Java code, depending on the connections and evaluations being compared. As a worst case, the continuous evaluation of a circuit whose highest-priority active rule is always changing is 144 times slower than Java code that implements the equivalent behavior. For the interactive programs that we have explored so far, most of the work is being performed in atoms that are implemented in Java, so SuperGlue's performance penalty is not very noticeable. On the other hand, if SuperGlue code is to be used in more computationally intensive ways, higher performance will be necessary. Given SuperGlue's lack of strong recursion, the direct effects of SuperGlue code on performance is linearly related to the number of rules in the program. However, as with cycles and termination, the presence of atoms implemented with arbitrary Java code make it difficult to reason about how overall program performance is affected by SuperGlue code.

6 Related Work

The signal abstraction originates from the functional-reactive programming (FRP) language Fran [10], which extends Haskell with signals. FRP is itself based on various synchronous data-flow languages such as Esterel [4], Lustre [6], and Signal [3]. More recent FRP languages include Fran's successor, Yampa [14], and FatherTime (FrTime) [8], which extends Scheme with signals. SuperGlue differs from Fran and Yampa and resembles FrTime in that it supports an asynchronous rather than synchronous concurrency model. By supporting an asynchronous concurrency model, SuperGlue code can easily integrate with imperative Java code, although we must deal with "glitches" that are described in Section 5. Unlike SuperGlue, both Fran and FrTime support signals with higher-order and recursively-defined functions. While the use of functions presents programmers with a well-understood functional programming model, function calls obscure the state-viewing relationships between components when compared to SuperGlue's connection-based programming model. Frappé [9] is an implementation of FRP for Java that is based on Fran. Because signals in Frappé are manipulated purely as Java objects, it suffers from the verbosity problems described in Section 2.

SuperGlue connections resemble simple constraints. Kaleidoscope [11, 12] supports general constraint variables whose values can be updated imperatively. Kaleidoscope constraints are more powerful than SuperGlue signals: its constraints are multi-way and it directly supports the expression of imperative code. However, these features also make Kaleidoscope more complicated than SuperGlue: the mixing of constraint and imperative constructs complicates the language, and the duration of its constraints must be explicitly programmed.

SuperGlue supports the expression of graph-like state with object-oriented abstractions. Inner objects with virtual refinement in SuperGlue resembles pattern nesting in BETA [17]. As described in Section 3.2, traits can be used to add new behavior to objects. As a result, SuperGlue's objects and traits are similar to the mixin [5], open classes [7], or views [21] that are used to add new methods to existing classes. SuperGlue's rules are similar to rules in logic programming languages such as Prolog [22], which query and prove logical relations. Although connections in SuperGlue are similar to simple relations, SuperGlue does not use first-order logic to manipulate connections. While logic programming focuses on expressing computations declaratively, SuperGlue focuses on the expression of glue code with as little expressiveness as possible. In this way, SuperGlue is more similar to SQL than Prolog.

Connections have long been used to describe dependencies between components– see various work on architecture description languages [16, 18, 20]. Visual tools in JavaBeans can be used to wire the bean properties of components together. ArchJava [1] is a language that uses the port-connection model to specify how components can interact in the implementation of a software architecture. ArchJava has a custom connector abstraction [2] that can be used to express a wide-variety of port types. In contrast, although SuperGlue only supports signals, it can support them with abstractions that cannot be easily expressed in ArchJava.

7 Conclusion

SuperGlue combines signal, object, and rule abstractions into a novel language for building interactive programs out of components. By focusing on glue code instead of component implementations, SuperGlue's abstractions sacrifice expressiveness so that glue code is easy to write. One consequence of this tradeoff is that SuperGlue does not have the control flow or recursion constructs found in general-purpose programming languages. The use of these constructs in glue code is often neither necessary nor desirable as they can often be replaced by rules that are more concise.

SuperGlue's design demonstrates how connections, an object system, and a declarative rule system can be effectively integrated together. The key to this integration is the use of object-oriented types to abstract over component connection graphs that are potentially unbounded in size. SuperGlue's support for connection prioritization via type extension and coercions via traits are all directly related to the use of types to abstract over connections.

7.1 Future Work

Although the features presented in this paper are complete, SuperGlue is still under development. We are currently refining our implementation of array signals, which were briefly described in Section 3.4. Additionally, we are improving component instantiation to be more dynamic. As presented in this paper, our language only supports a fixed number of components per program, which is too restrictive for many programs.

We have implemented SuperGlue with an initial prototype that is capable of running the case study described in Section 4. SuperGlue's implementation can be improved in two ways. First, compilation rather than interpretation can improve performance so SuperGlue can be used in computationally-intensive areas. Second, SuperGlue should be implemented in a way so that its programs can be developed interactively. This would allow the editing of a program's circuits while the program is running.

We plan to explore how SuperGlue can be used to build more kinds of interactive programs. We are currently investigating how a complete user-interface library in SuperGlue can enable user-interface programs that are more interactive. Beyond user interfaces, many programs can benefit from being more interactive than they currently are. For example, programming tools such as compilers are more useful when they are interactive. With the appropriate component libraries, SuperGlue would be a very good platform for developing these new kinds of interactive programs.

ACKNOWLEDGEMENTS

We thank Craig Chambers, Matthew Flatt, Gary Lindstrom, Gail Murphy, and the anonymous reviewers for comments on drafts of this paper and the preceeding research. Sean McDirmid and Wilson Hsieh were supported during this research in part by NSF CAREER award CCR–9876117. Wilson Hsieh is currently employed at Google.

References

1. J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in ArchJava. In *Proceedings of ECOOP*, volume 2374 of *Lecture Notes in Computer Science*, pages 334–367. Springer, 2002.
2. J. Aldrich, V. Sazawal, C. Chambers, and D. Notkin. Language support for connector abstractions. In *Proceedings of ECOOP*, Lecture Notes in Computer Science. Springer, 2003.
3. A. Benveniste, P. L. Geurnic, and C. Jacquemot. Synchronous programming with events and relations: the Signal language and its semantics. In *Science of Computer Programming*, 1991.
4. G. Berry. *The Foundations of Esterel*. MIT Press, 1998.
5. G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of of OOPSLA and ECOOP*, volume 25 (10) of *SIGPLAN Notices*, pages 303–311. ACM, 1990.
6. P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: a declarative language for programming synchronous systems. In *Proceedings of POPL*, 1987.
7. C. Clifton, G. T. Leavens, C. Chambers, and T. D. Millstein. Multijava: Modular open classes and symmetric multiple dispatch for Java. In *Proceedings of OOPSLA*, volume 35 (10) of *SIGPLAN Notices*, pages 130–145. ACM, 2000.
8. G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. To appear in ESOP, 2006.
9. A. Courtney. Frappé: Functional reactive programming in Java. In *Proceedings of PADL*, volume 1990 of *Lecture Notes in Computer Science*, pages 29–44. Springer, 2001.

10. C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of ICFP*, volume 32 (8) of *SIGPLAN Notices*, pages 263–273. ACM, 1997.
11. B. N. Freeman-Benson. Kaleidoscope: Mixing objects, constraints and imperative programming. In *Proceedings of of OOPSLA and ECOOP*, volume 25 (10) of *SIGPLAN Notices*, pages 77–88. ACM, 1990.
12. B. N. Freeman-Benson and A. Borning. Integrating constraints with an object-oriented language. In *Proceedings of ECOOP*, volume 615 of *Lecture Notes in Computer Science*, pages 268–286. Springer, 1992.
13. A. Goldberg and D. Robson. *SmallTalk-80: The Language and its Implementation*. Addison Wesley, Boston, MA, USA, 1983.
14. 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.
15. IBM. *The Eclipse Project*. http://www.eclipse.org/.
16. D. Luckham and J. Vera. An event-based architecture definition language. In *IEEE Transactions on Software Engineering*, volume 21, 1995.
17. O. L. Madsen and B. Moeller-Pedersen. Virtual classes - a powerful mechanism for object-oriented programming. In *Proceedings of OOPSLA*, pages 397–406, Oct. 1989.
18. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of ESEC*, 1995.
19. Y. Matsumoto. *Ruby: Programmers' Best Friend*. http://www.ruby-lang.org/en/.
20. N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of off-the-shelf components in C2-style architectures. In *Proceedings of ICSE*, pages 692–700. IEEE Computer Society, 1997.
21. M. Odersky and et. al. The scala language specification. Technical report, EPFL, Lausanne, Switzerland, 2004. http://scala.epfl.ch.
22. L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, Cambridge, MA, USA, 1986.
23. Sun Microsystems, Inc. *The JavaMail API*. http://java.sun.com/products/javamail/.
24. Sun Microsystems, Inc. *The Swing API*. http://java.sun.com/products/jfc/.
25. G. van Rossum and F. L. Drake. *The Python Language Reference Manual*, Sept. 2003. http://www.python.org/doc/current/ref/ref.html.