Turing Completeness Considered Harmful: Component Programming with a Simple Language

Sean McDirmid
École Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
sean.mcdirmid@epfl.ch

ABSTRACT

Languages are increasingly being used to reuse computations as components as well as to express these computations in the first place. However, the expression of computations and component assembly have opposing language requirements: the former requires flexible Turing-complete constructs that can express many interactions while the latter benefits from constructs that instead hide interactions. Accordingly, a simple non Turing-complete language should be used to express component assemblies rather than the general-purpose languages used to express computations. Unfortunately, simple languages often cannot express many kinds of useful component assemblies because they cannot abstract over "space" in the form of unbounded-size data structures or "time" in the form of mutable state.

We introduce SuperGlue as a simple language that can abstract over space and time without Turing-complete constructs. SuperGlue combines signals, which are values that can change over time, universally quantified rules, SQL-like arrays, and objects to express graphs of component connections that are unbounded in size and can change over time. With these constructs, SuperGlue is easy to use while also being capable of expressing useful programs such as user interfaces that view table-like or tree-like data. We have completed our initial design and implementation of SuperGlue and in this paper report on early experience.

1. INTRODUCTION

General-purpose programming languages provide programmers with a lot of flexibility because they are equivalent to Turing Machines in what they can express. However, this flexibility comes at the expense of complexity in program development. With recursion, indirect procedure calls, the ability to create aliases, and so on, programmers can express arbitrarily complicated forms of program control flow that are difficult to write down, debug, and otherwise reason about. Unfortunately, the complexity of a general-purpose language is necessary because it enables many kinds of computations that are needed in non-trivial programs.

Programmers not only use languages to express new computations, they also use languages to reuse existing computations in the form of components. Contemporary languages are increasingly judged on how well they support component reuse. The success of Java and C# is based significantly on their component libraries, while their language features such as garbage collection make it easier to integrate the components of these libraries. To improve their component-assembly capabilities, some general-purpose languages are also combined with component systems such as JavaBeans [32], COM [28], and CORBA [27]. A good example of such a combination occurs in Ruby [24], whose recent popularity is based on its ability to reuse components in the Ruby on Rails [14] framework.

Are the same Turing-complete constructs that are needed to express complicated computations also the best way to reuse computations that are encapsulated in components? We argue no: Turing-complete constructs are analogous to `goto` statements [11] in how they obscure component relationships. Instead, it would be better to express component assemblies in a **simple language**, meaning a non-Turing complete language, where component relationships are obvious and unobscured. Although very common, simple languages tend not to get as much attention as more powerful general-purpose languages. Examples of simple languages include linking languages such as Jiazzi [25], which is used to assemble Java modules together. Many simple languages are based on XML; e.g., XML is used in Eclipse [18] to express how plug-ins are integrated together. The point of using a simple language to assemble components together is that component relationships are necessarily obvious; e.g., it is impossible to bury a relationship inside a sequence of recursive procedure calls. This in turn makes simple languages easier to use with respect to programming effort.

Unfortunately, simple languages are often not effective at component assembly because they cannot expressively abstract over **space** in the form of unbounded-sized data structures or **time** in the form of mutable state. Abstracting over space is important because components must often communicate data structures of unbounded sizes such as lists, tables, or trees. For example, configuring a user-interface tree view requires describing its model as a tree in component-assembly code. In a general-purpose language, control-flow constructs can always be used to iterate over these data structures, whereas direct iteration is inconvenient in a simple language. Abstracting over time is important because components must often communicate mutable state. For

example, component-assembly code must ensure that a user-interface tree view is updated when new nodes are added to or removed from its model. In a general-purpose language, control-flow constructs can be used to manage aliasing, event handling, or polling between components, whereas managing such details is inconvenient in a simple language.

This paper shows how a simple language can expressively abstract over space and time without resorting to the Turing-complete constructs of a general-purpose language. Our approach is based on the following two mechanisms:

- **Universal quantification**, which allows one variable to abstract over an unbounded number of values. With universal quantification, a finite amount of declarative component-assembly code can generate an unbounded number of connections between components. Although universal quantification is often noted for its use in logic programming languages such as Prolog [31], it is also used in simple relational languages such as SQL to abstract over tables of unbounded sizes. Additionally, universal quantification is often used to express rules in simple languages; e.g., build rules in GNU Make [30] can be applied to an unbounded number of code resources.

- **Signals**, which act as declarative representations of state by encapsulating values that can change over time. Although signal terminology originates from research in functional reactive programming [12, 17, 9], signals also resemble the live data-flow values that are used in various simple visual languages such as Quartz Composer [21]. When components are assembled together through signals, they can communicate changes in their state without involving component-assembly code in the details of this communication.

We combine universal quantification and signals in our language SuperGlue, which is designed specifically to express component-assembly code. SuperGlue components, which can be implemented in Java, communicate through externally connected signals. SuperGlue code is then expressed as rules that build a graph of signal connections between the components of a program. Through our use of universal quantification, this graph and the number of components in a program are unbounded in size. SuperGlue is novel in its approach for organizing connection rules with object-oriented abstractions. SuperGlue components appear to assembly code as bundles of nested objects that import and export signals. Object-oriented extension is used to organize these rules and enable a form of connection overriding. Components can be described with traits that act to separate interface from implementation and allow rules to adapt incompatible components. Besides its use in rules, universal quantification is also used to express arrays of signals, where navigation within these arrays occurs in a SQL-like way. Through its combination of signals, objects, rules, and universal quantification, SuperGlue is the first simple language that we are aware of that can abstract over both unbounded-sized data structures and mutable state.

Our previous work [26] describes how SuperGlue combines signals and objects together. The work in this paper puts that work into context to show that the power of Turing-complete languages is not necessary, and can even be harmful, in expressing component compositions. Additionally, this paper provides a comprehensive introduction to SuperGlue's more experimental and previously unreported constructs, such as the use of universal quantification to represent arrays of values.

The rest of this paper is organized as follows. Section 2 looks at how existing general-purpose and simple languages are used in component assembly. Section 3 uses multiple examples to describe SuperGlue. Section 4 provides a critical discussion of SuperGlue's ease of use and expressiveness. Section 5 describes our prototype implementation of SuperGlue. Section 6 presents related research while Section 7 presents our conclusions and directions for future work.

2. BACKGROUND

Component assembly is easier if we can reduce the amount of explicit communication that must occur between components. Explicit inter-component communication is reduced in general-purpose languages in a variety of ways. Garbage collection eliminates a lot of the code needed to manage memory between components. Introspection techniques can also be used to automatically infer certain component interactions. For example, in Ruby on Rails [14], a class definition can express a method using a naming convention whose implementation is auto-generated to access certain data in a relational database.

The best opportunity for reducing inter-component communication in general-purpose languages has been and remains through the use of standard component interfaces. As an example, the adoption of a standard observable list interface could significantly reduce the amount of assembly code needed to build user-interfaces in Java [1]. Such an interface could be used to describe both a list of rows in a Swing [34] user-interface table as well as a list of email messages in a JavaMail [33] email folder. Assembling these components together would then be much easier than the current situation: table rows and email messages are provided through unique list interfaces that are incompatible in both method names and the event-handling used to communicate changes in the lists. Instead of just writing down the Java statement `table.setRows(folder.getEmailMessages())`, adapting the row and message list interfaces requires writing down a lot more code and more importantly, spending a lot more time debugging this code.

Standard interfaces are sometimes avoided because they can prevent components from being used in performance-critical situations. For example, given performance concerns, an email-message addition event in JavaMail is discontinuous and unordered, while a table row addition event in Swing is continuous and ordered. Scripting languages, such as Python [36], Ruby [24], and Perl [38], focus on ease of use over performance and so are more aggressive in their use of standard interfaces. As an example, a standard pipe interface is provided by many scripting languages to facilitate component communication through character streams. In Bourne Shell [7], the code `cat status.txt | grep urgent | wc` assembles three components together using pipes (`|`) to output the number of lines that contain "urgent" in the file `status.txt`. Such a program is significantly easier to write and debug with pipes than without.

We refer to a component relationship that can be encoded without inter-component communication as a *declarative connection*. Declarative connections can be expressed in general-purpose languages through standard interfaces

```
<action id="com.acme.actions.Cleanup"
 icon="icons/cleanup.gif"
 tooltip="Clean up markers in selected resource"
 class="com.acme.actions.RunCleanup"
 enablesFor="1">
<enablement>
<and>
 <objectClass
  name="org.eclipse.core.resources.IFile"/>
 <objectState name="extension" value="java"/>
</and>
</enablement>
</action>
```

Figure 1: Part of the XML code that configures an Eclipse plug-in.

such as an observable list or pipe interface. Unfortunately, general-purpose languages are limited in how they can support declarative connections. Because these languages do not directly support declarative programming, declarative connections are often obscured by being expressed as procedure calls. These procedure calls can occur at any discrete time point in the program's executions, which enables flexibility in managing declarative connections but sacrifices their ease-of-use benefits. For example, controlling a declarative connection according to some other state in Java requires installing and uninstalling the connection in an event handler.

2.1 Simple Languages

Simple (non-Turing Complete) languages come in many forms. In compiler construction, scanning and parsing can be performed with Lex and Yacc [20], which are respectively based on finite-state and push-down automata models of computations. Other simple languages focus primarily on expressing connections. As an example, consider the XML code fragment in Figure 2.1 that configures an action that is provided by an Eclipse plug-in. Most of this code specifies the action's configuration options, such as its label, icon, and a Java class that implement the action's invoke behavior. These are examples of declarative connections that express how the plug-in is connected to the rest of the IDE.

The expressiveness of a simple language is often limited in ways that are very noticeable to programmers. At the bottom of Figure 2.1, relationships are expressed to enable a plug-in action only when a Java file is selected. More complicated behavior is difficult or impossible to express in XML code; e.g., it is impossible to express that an action should only be enabled when one Java and one XML file is selected. The Java code of a plug-in can be used to express more complicated behavior, but this code will only execute after the plug-in has been loaded. This situation can (and does) lead to some significant inconsistencies in Eclipse; e.g., the user can trigger an enabled action that does not execute because freshly-loaded Java code decides the action should not have been enabled. If simple languages are to be more effective as component-assembly languages, they need to become more expressive so they can avoid these situations.

2.2 Universal Quantification and Rules

The expressiveness of a simple language can be improved with universal quantification. A single universally-quantified variable can abstract over a set of values that is determined by the conditions expressed over the variable. The most well-known example of a simple language that supports universal quantification is SQL. Code in SQL is expressed as queries that operate over tables of data from databases. For example, the SQL query `SELECT name FROM students WHERE gpa > 3` computes a column of names from a students table for all rows with GPA columns greater than three. The use of universal quantification in SQL provides a very powerful way of manipulating data without resorting to Turing complete constructs. Unfortunately, SQL is designed only as query language, and either an extension or another language (both of which are often general-purpose) are needed to use the results of its queries.

Universal quantification can be combined with rule constructs to both express queries and process query results. The antecedents of a rule express queries that restrict how the rule's universally-quantified variables can be bound. The consequents of a rule then refer to these variables to operate on the resulting values of the query. An example of a simple language that supports rules is Make [30], where implicit build rules direct resource translation in a build. The following implicit build rule specifies how C files are translated into object files:

```
.o.c:
   $(CC) $(CFLAGS) -c $*.c
```

The antecedent of this build rule is `.o.c`, which means the translation of some file with extension `.c` to a file with the same base name but with extension `.o` is needed. The universally quantified variable of this build rule is `$*`, which is the base name of the file. The consequent then compiles the C file (`$*.c`). This implicit build rule is automatically applied by the run-time whenever a `.o` file is requested and a `.c` file with the same base name exists.

Universal quantification and rule-based reasoning is characteristic of logic programming languages, such as Prolog [31], which are often Turing complete. Rule application in a logic language is managed by an automated proof engine. In a logic that supports transitive proof relationships between facts, Turing completeness comes from the proof engine's recursive application of rules to traverse unbounded data structures such as lists or trees. Although data structure traversal is performed by the proof engine, traversal can be controlled by the programmer through antecedent ordering, which is often necessary in Prolog to achieve acceptable performance and avoid infinite query loops. Exposure to proof engine details prevent logic programming languages from having the ease-of-use qualities of simple languages.

Constructs that allow languages to abstract over multi-element data without loops or recursion are often added to general-purpose languages to enhance their ease of use. For example, C# is being enhanced with direct support for expressing SQL queries [15]. Array programming languages such as APL [13] can operate directly on arrays of values. Functional programming languages often support some form of *list comprehension* [37] that hides list traversal logic from programmers. The primary limitations of these constructs is that they cannot be used implicitly, i.e., they must be invoked within the program's control flow, and can only operate on concrete sets of data. Rule engines can get around these limitations; e.g., Rake [39] enables the expression of implicit build rules in Ruby. However, such extensions lead

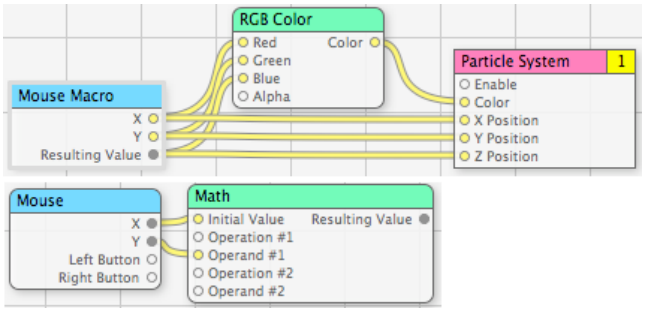

Figure 2: An example of a program in Quartz Composer (top) and the composition of the Mouse Macro patch (bottom).

to a mixture of declarative and non-declarative code where two languages are effectively being used at once.

2.3 Signals

Another way to improve the expressiveness of a simple language is to enhance it with support for reasoning about state. State is often associated with imperative programming where discrete reads (peek) and updates (poke) are expressed in the control flow of the program. However, imperative updates are often associated with complicated computations, and therefore are better encapsulated inside components. A simple languages should instead focus on the communication of changes in state between components, as well as the transformation and combination of state.

In a general-purpose language, the transparent communication of changes in state can be achieved to some degree through standard interfaces; e.g., through a standard observable list interface. The transparent communication of state can be better achieved in a simple language through *signals*, which are component connectors designed specifically to encapsulate changes in state. Signals are more powerful than standard interfaces because they can be easily transformed or combined. For example, consider the expression `z = x + y`: if `x` and `y` are signals, then `z` is also a signal whose current value is always the sum of the current values of `x` and `y`. Any change in the value of `x` or `y` will cause the value of `z` to immediately change.

Examples of signal-like constructs can be found in Quartz Composer [21], which is a visual programming languages that supports the construction of interactive animations. Components in Quartz Composer are "patches" that input and output signal-like live data-flow values. An example of a Quartz Composer program is shown in Figure 2. The `Mouse Macro` patch outputs the current horizontal (`X`) and vertical (`Y`) values of the mouse pointer, as well as the sum modulo (`(X + Y) % 1.0`) of these values (`Resulting Value`). The latter output is an example of how state can be combined and transformed with signals. These three values are then used to compute the RGB color and three-dimensional position of a particle system. The resulting program creates a particle system whose position and color changes as the mouse is moved around the computer screen.

Signals enable the communication of state between components without the expression of event handling or polling in glue code. As a result, writing and debugging a program is easier; e.g., there is no complicated event handling to debug, only connections to express and observe. Besides being constructs in visual languages, signals can also be constructs in textual general-purpose programming languages. Research on functional reactive programming [12, 17, 9], where the signal term originates, demonstrates how signals can be added to pure functional programming languages. Although the incorporation of state into a general-purpose language can significantly ease programming, programmers must still depend on control and data flow constructs to write programs. We revisit research on functional-reactive programming as related work in Section 6.

3. SUPERGLUE

Although universal quantification and signals have already been shown to make simple languages more expressive, they have not previously been combined in a simple language that can abstract both over space (unbounded data structures) and time (mutable state). SQL and Make [30] can abstract over unbounded data structures but have no way of reasoning about mutable state. Quartz Composer [21] can abstract over mutable state but does not support abstraction over data structures very well; e.g., multiplexor-like components are used to deal with iteration in a very visual but also very tedious way. Without the ability to abstract over both space and time, simple languages are often severely constrained and do not feel like real programming languages.

SuperGlue is a textual simple language that combines signals with universally-quantified rule constructs. These mechanisms are organized in an object-oriented way: components in SuperGlue appear as bundles of nested objects that import and export signals. As in other object-oriented languages, SuperGlue's objects are created from classes. As an example, the following code declares the `Bird` class, creates the *tweety* bird object, and connects Tweety's imported `canFly` signal to `true`.

```
class Bird
{ import canFly :  Boolean;
  export strength :  Int; }

let tweety = Bird;
tweety.canFly = true;
```

Connections are expressed using assignment syntax, where the signal being connected to is on the left (*tweety*`.canFly`) and the expression being connected from is on the right (`true`). We use assignment syntax because signal connections can be thought of assignments that are evaluated continuously; e.g., in the above code and Tweety can always fly.

To make our example more interesting, the following code expresses that Tweety can only fly if his strength is greater than `100`:

```
if (tweety.strength > 100) tweety.canFly = true;
else tweety.canFly = false;
```

A connection in SuperGlue is the consequent of a rule whose application can be restricted by the rule's antecedents, which are expressed inside as conditions in `if` statements. In the above code, the connection of `true` to Tweety's `canFly` signal is applicable only when Tweety's strength is at a certain

level. Rules that connect the same signal can be explicitly prioritized with `else` clauses, which apply containing rules only when corresponding `if` conditions are false. For example, because of the `else` clause, Tweety cannot fly whenever his strength is below a certain level. Because the value of a signal can change during a program's execution, whether a rule's antecedents are true and the rule can be applied can also change. For example, Tweety can fly again after it regains strength. Rules that can connect to the same signal are organized into *circuits* that resemble data-flow graphs. At run-time, the antecedents of these rules are continuously evaluated, and changes in their values will switch how a signal is connected. In our example, the circuit for Tweety's `canFly` signal switches the signal's value between `true` and `false` depending on the current value of Tweety's `strength` signal.

Rules can express how more than one object is connected through universally-quantified variables. As an example, the following code connects the `canFly` signal of every `Bird` object:

```
var bird :  Bird;
bird.canFly = true;
```

Variables declared with the `var` keyword abstract over every value compatible with their specified type; e.g., the `bird` variable abstracts over all `Bird` objects.

Besides being explicitly prioritized with `else` statements, rules are also implicitly prioritized by the specificity of the types that they are expressed over, where specificity is determined by object-oriented extension. Extension relationships are expressed in a standard object-oriented way; e.g., `class Penguin extends Bird` declares a `Penguin` class whose objects are also `Bird` objects. Rules that connect signals in all penguin objects are less general and so have a higher priority than rules that connect signals in all bird objects. For example, the following code prevents penguins from flying although most other birds can:

```
var penguin :  Penguin;
penguin.canFly = false;
```

Because implicit rule prioritization via object-oriented extension relationships enables a form of connection overriding, object-oriented abstractions complement rules very effectively in SuperGlue.

3.1 Inner Classes

Although variables in SuperGlue can abstract over all objects of a class, there is still not much to abstract over as these objects can only contain a finite number of signals. To solve this problem, objects in SuperGlue can be nested, where inner objects act to extend the interface of their containing object. For example, inner node objects can be nested in an user-interface tree view object to describe the hierarchical structure of the tree. The creation and identity of an inner object is hidden from component-assembly code, allowing component implementations to create inner objects only when they want to import or exports additional signals. As a result, inner objects allow a component to import or export an unbounded number of signals through a bundle of nested objects, allowing the component to use or provide data structures of unbounded size, such as trees or tables.

```
class GraphView {
  inner Node
  { import edgeA, edgeB : Node; }
  import root : Node;
}
class ProcedureModel {
  inner Statement
  { export branch : Statement; }
  inner Conditional extends Statement
  { export otherwise : Statement; }
    export entry : Statement;
}
```

Figure 3: Declarations of SuperGlue classes that are used to create user-interface graph view objects and procedure model objects.

Inner objects are described as inner classes whose declarations are nested in other top-level or inner class declarations. As an example, the `GraphView` and `Program` classes that are declared in Figure 3.1 use inner classes to describe unbounded hierarchies of nested objects. The `GraphView` class nests an inner `Node` class, which describes the nodes of a graph, while the `ProcedureModel` class nests an inner `Statement` class, which describe the statements of a procedure. Inner classes are used as signal types within the classes that they are nested in. For example, the `Node` inner class nested within `GraphView` is used as the type for the `root` signal of the `GraphView` class as well as the type for the `edgeA` and `edgeB` signals that are declared in `Node` itself. Inner classes can also be extended by other inner classes. For example, the `Statement` inner class nested within `ProcedureModel` is extended by the `Conditional` inner class.

Initially, a graph view and procedure object can be assembled together by connecting their `root` and `entry` signals:

```
let graph = GraphView;
let proc = ProcedureModel;
graph.root = proc.entry;
```

As a result of this connection, a `Statement` object nested in the top-level *proc* object is connected to a `Node` object nested in the top-level *graph* object. In SuperGlue, connecting an expression to an imported signal does not replace the imported signal with the expression. Because no replacement occurs, the type of the expression connected to an imported signal does not need to, and often does not, match the declared type of the imported signal. In our example, a graph node object can be connected to a statement object even though the `GraphView.Node` and `ProcedureModel.Statement` inner classes are unrelated in their types. Instead of replacement, each connection has a run-time representation between the imported signal's internal value, whose type is indicated by the imported signal's type, and the value of the expression, whose type is indicated by the exported signal's type. The connection can then be examined at run-time in a *connection query*, whose result can be used to resolve any incompatibilities between the values involved in the connection. Following our example, the following code uses variables and a connection query to abstract over every connection from a statement object to a node object:

```
var node :  GraphView.Node;
```

```
trait List {
  inner T;
  port size   : Int;
  port index  : Int[];
  port item   : T[];
}
trait TableTree {
  inner Column;
  inner Row {
    port columns : List(T : Column);
    port children : List(T : Row);
  }
  port rows : List(T : Row);
}
trait Titled { port title  : String; }
trait Cell   { port widget : Widget; }

class TableTreeView imports Table {
  override Column imports Cell, Titled;
  export selection : List(T : Row);
}
```

Figure 4: SuperGlue traits that describe lists, table trees, and titled entities; and a class for creating table tree view objects.

```
var stmnt :  ProcedureModel.Statement;
if (node = stmnt)  node.edgeA = stmnt.branch;
```

A connection query is expressed using connection syntax inside an `if` condition (e.g., `if (stmnt = node) ...`). When an unbound variable is specified on a connection query's right, the query is true if the evaluation of its left expression is or is connected to a value whose type is compatible with the variable's type, in which case this value is bound to the variable. If the connection query in our example is true, the `stmnt` variable is bound to the statement object that is connected to the node object bound to the `node` variable. The bound `stmnt` variable is then used to connect the `stmnt` object's `branch` signal to the node object's `edgeA` signal. Code of a similar structure can deal with conditional statements:

```
var cond :  ProcedureModel.Conditional;
if (node = cond)  node.edgeB = cond.otherwise;
```

As a result of this code, the graph view will contain as a node every statement of the procedure model.

It is up to component implementations to decide if inner objects have their own identities and state. Internally, `GraphView` components identify node objects by the objects that are connected to them. As a result, connecting the same object to different node objects acts to identify the same node (i.e., the same state) within the graph, making it easy to express cyclic graphs. In our example, if the procedure model object exports the same statement object through different signals, then these statements are associated with the same node in the graph view. Because the identity of an inner object is configured by its enclosing object, inner classes cannot be explicitly instantiated from outside their containers. Instead, the creation of an inner object is a consequence of a connection, where the inner object acts to describe part of the connection.

3.2 Traits

Because connections in SuperGlue have run-time representations and are not supported with replacement semantics, class types and types declared in classes cannot be shared between different classes. Instead, the only types that can be shared between classes are *traits*, which describe the signals and inner objects of an object without implying specific implementations or the import/export polarities of their signals. Standard interfaces, which are as important in SuperGlue as they are in a general-purpose language, are expressed in SuperGlue as traits. The data-describing `List` and `TableTree` traits and user-interface `Titled` and `Cell` traits are declared in Figure 3.2. Syntactically, trait declarations resemble class declarations, except that signals are declared with the `port` keyword rather than with the `import` or `export` keywords.

A class can implement a trait by either importing or exporting its members. The `TableTreeView` class declared in Figure 3.2 imports the members of the `TableTree` trait by using the `imports` keyword. Inner classes, which can be declared inside a trait, are inherited by classes that implement the trait. For example, the `TableTreeView` class inherits the `Column` and `Row` inner classes. Inherited inner classes can be overridden with the `override` keyword, where overriding involves having the inner class implement additional traits or signals. For example, the `TableTreeView` class overrides the `Column` inner class to enhance it with imported user-interface `Titled` and `Cell` traits. By implementing traits and overriding their inner classes, traits can be effectively combined to describe complicated component interfaces; e.g., data-describing and user-interface traits can be combined to describe the table tree view class.

For the same reason that class types cannot be shared between classes, SuperGlue does not support parametric polymorphism. Instead, overriding is used to perform inner class type refinement. In Figure 3.2, the item type of a list is described by the inner class `T` that is nested in the `List` trait. To specify an item type, the `T` inner class must be overridden to extend or implement the specific inner class or trait type. This can concisely be expressed with parentheses and colon syntax when using the `List` trait as a signal type, e.g.,

```
export selection :  List(T :  Row)
```

is syntactic sugar for:

```
inner selection$ exports List
{ override T extends Row; }
export selection :  selection$;
```

Given the frequent use of traits to describe signal types, this form of syntactic sugar is essential as opposed to just a convenience.

Connections involving the same traits are compatible and do not require additional rules to connect their sub-signals. As an example, the following connection from a list of expressions to a list of rows does not require extra rules to connect their `List` signals:

```
class ProcedureModel {
 inner Expression {
  export opcode :  Op;
  export lhs :  Register;
  export operands :  List(T :  Register);
 }
```

```
  inner Statement
  { export expressions :  List(T :  Expression); }
}
let exprVw = TableTreeView;
/* stmnt is bound to some statement object */
exprVw.rows = stmnt.expressions;
```

However, additional rules are still be needed to address the incompatibility between the `Row` and `Expression` item types.

3.3 Arrays

Signals whose declared types are followed by brackets are *arrays* that can be associated with multiple values at the same time. The `List` trait declared in Figure 3.2 declares two array signals: an `index` array signal that is always associated with all integers, and an `item` array signal that is associated with the items of a list object. The `size` signal of the `List` trait is not an array and is always associated with the singular size of a list object. The value of an array signal can be explicitly narrowed in a SQL-like way, which can cause peer array signals to become narrowed implicitly. For example, explicitly narrowing the `index` signal of a list will implicitly narrow its `item` signal, enabling access to a list item by its index. For example, the following de-sugared code narrows the `index` signal of a graph node selection to the value `0`:

```
class GraphView {
   ...  export selection :  List(T :  Node);
}
select item from graph.selection where index = 0;
```

The result of this expression is whatever the first selected node is, or an *IndexOutOfBounds* exception if no nodes are selected. Because narrowing is such a common operation, syntactic sugar is always used to express it more concisely. The following sugared code is equivalent to the above SQL-like code:

```
graph.selection(index = 0).item
```

Parentheses that follow a signal expression contain the narrowing expression. The result of the narrowing is then the value of the signal expression.

Because signals are values that change over time, array signals can abstract over both time and space. In our example, the value of the first selected node of a graph will change as the user interactively selects and deselects graph nodes. Such dynamic behavior can be used to easily establish a master-detail relationship from the graph view (master) to a table tree view of a statement's expressions (detail):

```
let exprVw = TableTreeView;
if (graph.selection.size >= 1 &&
   graph.selection(index = 0).item = stmnt)
  exprVw.rows = stmnt.expressions;
```

As a result of this SuperGlue code, the expressions of the expression view belong to the first statement that is selected in the graph.

Besides selection of list's item by index, containment and index lookup operations are supported by lists without separate `contains` and `indexOf` signals. As an example consider the following statements:

```
/* statement stmnt0 contain expression e0?  */
if (stmnt0.expressions = e0) ...;
/* index (n) of expression in statement */
let n = stmnt0.expressions(item = e0).index;
```

Unlike an `indexOf()` list method in Java, which returns only the first index of an item, the above statement will bind `n` to an array (possibly empty) that contains each index in the list where `e0` is an item. SuperGlue array signals directly support the filter and map forms of list comprehension. As an example, consider the following statements:

```
/* filter for expressions whose lhs are v0 */
let x = stmnt0.expressions(item.lhs = v0);
/* convert filtered expressions into a list */
let list0 = MkList(input = x);
/* list of opcodes for filtered expressions */
let list1 = MkList(input = x.item.opcode);
```

Array signals and connection queries interact in one very notable way: when the left side of a connection query evaluates to an array value, the connection query binds a filtered array value to the right side variable. Such a connection query will always evaluate to true and allows us to represent column types directly as the row members that will fill out a row's columns. As an example, consider the following code that specifies the expression members to display as columns in the expression view:

```
var row :  exprVw.Row;
var expr :  Statement.Expression;
if (row = expr)
 row.item.columns = expr.[lhs, opcode, operands];
```

The syntax `e.[s0, s1, s2]` creates a list by accessing the `s0`, `s1`, and `s2` signals from expression `e`. When querying column headers, the `row` variable is bound to every row of the table. If the table is empty (no statement is selected in the graph), then the correct headers are still displayed because the above connection query is still true. The implementation of the table view class expects a column header to be singular, meaning they are arrays of exactly one value. Singularity requirements cannot be enforced statically and are checked dynamically. As a result, if different rows of a table view have different columns, an ambiguous connection exception will be propagated at run-time.

3.4 Adaptation

By using variables whose types are traits as the subjects and targets of connections queries, connections can be reasoned about generically without referring to specific classes. As an example, the following code defines how the LHS of an expression should be titled in any case where the subject is connected to an object of the `Titled` trait declared in Figure 3.2:

```
var ttld :  Titled;
if (ttld = expr.lhs) ttld.title = "Result";
```

This connection rule will title expression LHS properties in

```
var view : HasView;
var proc : ProcedureModel;
if (view = proc) {
  let graph = Graph;
  let  exprVw = TableTreeView;
  ... /* configure sub-views. */
  view.panel =
    SplitPane(left = graph, right = exprVw);
  );
}
```

Figure 5: A fragment of SuperGlue code that defines a view for a procedure model.

many user-interface contexts; e.g., as the title of a table view column, form view field, and so on.

3.5 Instantiation

Because SuperGlue is declarative and lacks control flow, when a class is instantiated into an object is not explicitly specified. In our examples so far, this has not been much of a problem if we assume class instantiation occurs when the program starts. However, this assumption requires that only a fixed number of top-level (non-inner) objects are created in a program, which is unrealistic for non-trivial programs. Instead, top-level objects are associated with a container object based on how they are used in connections. If a top-level object or its signals are connected to the signals of exactly one other object (top-level or inner), then the former object is contained in the latter object. If the top-level object is connected to the signals of multiple objects, then the object is contained in a container object common to these objects, if such an object exists. A top-level object with a container is lazily instantiated once per container, while a top-level object without a container is lazily instantiated once per program. As an example, consider how the code in Figure 3.5 defines a view for a procedure model. The *graph* and *exprVw* objects belong to the SplitPane object because these objects are connected to the split pane's left and right signals. The SplitPane object in turn is contained in some top-level or inner object that is bound to the view variable at run-time. As a result of these relationships, separate sets of *graph*, *exprVw*, and SplitPane objects with their own state are instantiated to create multiple views for different procedure models that can be displayed at the same time.

4. DISCUSSION

SuperGlue was designed from the beginning for component assembly in the mold of a module linking language. Early on, we realized that there were two very different paths to choose in SuperGlue's design: the path of expressiveness, which involves adding constructs such as recursive functions, and the path of simplicity, which involves remaining faithful to a connection-based paradigm. Each path has its disadvantages: the path of expressiveness entails complexity as well constructs that did not fit very well with connections, while the path of simplicity entails not being able to express many kinds of programs. Perhaps because programmers do not like being restricted, the path of expressiveness is often considered as being more viable; e.g., ArchJava [2], which we discuss in Section 6, focuses on managing complexity rather than restricting it. However, the path of simplicity has its own merits and its restrictions are more acceptable when many complex computations are encapsulated within components. For these reasons, we chose to explore the path of simplicity in SuperGlue's design.

The effectiveness of SuperGlue's design can be judged according to two criteria. The first criteria is ease of use, which is related to the amount of time a programmer must spend writing and debugging code. SuperGlue is not Turing complete, but this does not automatically mean that it is easy to use. The second criteria is expressiveness, which determines how often SuperGlue can be used to write a program. SuperGlue will never be as expressive as a general-purpose language, but it should be generally applicable: SuperGlue's breadth of expressible programs should be comparable to a general-purpose language, although its depth will certainly be more limited.

One way to reason about SuperGlue's ease of use is through code structure metrics. Adjusting results for verbosity to Java's benefit, an initial case study reported in previous work [26] shows that the SuperGlue code of an email client is about half the size as the Java code for a client with similar features. We expect that this reduction will improve with better components and the addition of more advanced program features. Such reduction is necessary: we estimate that at least a factor of four reduction is needed before SuperGlue can be considered significantly easier to use. We have yet to measure debugging effort, which will require user testing. Given SuperGlue code's lack of control-flow, debugging should focus on observing an executing program's overall behavior rather than trying to reason about what the computer is doing. On the other hand, early experience indicates that debugging in SuperGlue also involves probing signal values and visualizing their connecting circuits. Finally, we must consider SuperGlue's "weirdness factor." Most programmers are used to languages that model the computer (imperative), first-order logic (logic), or the lambda calculus (functional). SuperGlue's model is like none of these, and so it is often outside of a programmer's existing comfort zone.

To improve expressiveness, SuperGlue's design tries to consider component-assembly tasks that programmers are likely to perform. However, comprehensively defining legitimate component-assembly tasks is almost impossible because they are not well-defined in existing languages. As a result, we started looking at user-interface programs whose components are fairly explicit: widgets on the screen are components that are assembled with components that model data. Unfortunately, our focus on user-interface programs biases SuperGlue's component model: components are encapsulated containers of state that interact by viewing each others' state. The applicability of SuperGlue's component model outside of user-interface programs has yet to be explored in depth. We argue that programs that transform state extend beyond user-interfaces to more general data-processing domains. However, given our current lack of experience in these domains, we see user-interface programs as a starting point for component assemblies that programmers are interested in writing.

Although simple (non-recursive) computations can be expressed in SuperGlue, complicated (recursive) computations must be encapsulated inside components that are implemented in a general-purpose language. Unfortunately, the use of a general-purpose language does not provide an easy out to SuperGlue's expressibility problems: the code needed

to bridge between a general-purpose language and SuperGlue is complicated and only worthwhile in components that are reused multiple times. For this reason, a couple of constructs allow some complicated computations to be expressed in SuperGlue code. SuperGlue supports stream constructs that are used to reason about discrete events and commands, e.g., the following code deletes email messages selected in a table view when a button is pushed:

```
on (button.pushed)
 if (msgVw.selection.item = msg)    do msg.delete;
```

Also various generic low-level components can be assembled to express computations such as zipping; e.g., the following code sums element-wise two lists of integers together:

```
let sum = Zip(input0 = list0 &&  input1 = list1 &&
 result =  list0.item + list1.item);
```

This kind of SuperGlue code is not significantly easier to develop than equivalent constructs in a general-purpose language. Their only benefit is that they avoid the need to write general-purpose code, which would otherwise discourage many programs from being written in SuperGlue.

Related to ease of use and expressiveness is the integrity of a SuperGlue program. For expressiveness reasons, type checking in SuperGlue occurs dynamically: a connection can fail at run-time with either an unconnected or ambiguous (non-singular) connection error. Given the inability of SuperGlue code to deal with control flow, type checking and other errors in SuperGlue are propagated as values for the affected signal. As a result, a programmer can only learn about errors by probing signals, while components must be implemented to gracefully deal with being connected incorrectly.

Another integrity issue to consider is that of termination. Because SuperGlue is not Turing complete, there is no way for SuperGlue code by itself to cause an infinite loop. However, the components that are assembled by SuperGlue code can loop indefinitely depending on how they are connected and implemented. For example, a user-interface tree that only accesses a node's children when the node is expanded by the user will not hang the program even if the tree being viewed is of an infinite depth. On the other hand, if the tree is configured so that all nodes are initially expanded, non-termination can occur on an infinite-depth tree. Cycles between circuits can easily cause non-termination problems. Cycles that cannot be trivially detected occur because of encapsulated dependencies between a component's own imports and exports; e.g., connecting the exported selected rows signal of a user-interface table to the table's imported rows signal would cause an infinite loop. Statically detecting and rejecting such cyclic connections appears feasible and will be explored in future work.

5. IMPLEMENTATION

Our prototype of SuperGlue, which is actually our second, implements the features described in this paper. The prototype organizes rules that connect an object's signals into circuits using object-oriented extension relationships, discussed at the beginning of Section 3, to prioritize these rules. A circuit is then transformed into a data-flow graph that can be traversed at run-time. Traversal of these data-flow graphs is augmented with additional mechanisms that enable the narrowing of array signals and instantiation of object cliques. A detailed discussion of these mechanisms is beyond the scope of this paper.

We have found that implementing SuperGlue's support for signals is more difficult than it first appears. The change of a signal's value must be propagated in such a way that avoids various kinds evaluation inconsistencies known as *glitches* [10], which can result in problems such as race conditions. Glitches occur because a change in one signal can cause other signals to change, and if these changes are propagated immediately, circuit evaluations may become inconsistent. To avoid glitches, we serialize all signal changes in our current prototype, which is admittedly Draconian and we hope to change in future prototypes.

As for performance, according to simple micro-benchmarks, our current very unoptimized prototype executes SuperGlue code around 100 times slower than a JVM executes Java code. However, computations that dominate program execution are often encapsulated in components that are implemented in Java. Additionally, the programs we have looked at so far with SuperGlue are mostly user-interface intensive, where SuperGlue's performance penalty is seven less significant. A useful analysis of SuperGlue's performance will require examining programs where SuperGlue code execution is more significant.

Although technically not part of a SuperGlue implementation, component implementations represent a significant amount of complexity in our implementation. While our prototype is implemented in around 4,000 lines of Java code, we have written a similar amount of Java code to adapt existing Java components as SuperGlue components. Java code interfaces with SuperGlue code according to signal-representing interfaces that resemble the following Java interface:

```
interface Signal {
 Value current();
 void install(ChangeObserver o);
 void uninstall(ChangeObserver o);
}
```

Although forcing Java code to communicate through these interfaces simplifies glue code, it necessarily complicates component implementations. For example, adapting Swing's [34] `JTable` and `JTree` components into SuperGlue components respectively require around 300 and 400 lines of Java code. As a result, the integration of code between Java and SuperGlue is not very seamless and constitutes a serious programming effort.

6. RELATED WORK

As far as we know, SuperGlue's design strategy is unique in enhancing a simple language to be more expressive without becoming a general-purpose language. This probably has something to do with the "not real languages" status often affixed to simple languages, in spite of their ubiquity. Research related that we have found involve going in the opposite direction; i.e., enhancing general-purpose languages with simple declarative constructs to improve ease of use.

SuperGlue's signals originate from research on functional-reactive programming (FRP), which enhance pure functional languages with declarative signals constructs to ease reason-

ing about state. FRP in turn is inspired by various synchronous data-flow languages such as Esterel [5], Lustre [8], and Signal [4]. Haskell FRP systems include Fran [12] and Yampa [17], which enhance Haskell with signals that depend on an explicit notion of time that is synchronous and continuous. In contrast, FrTime [9] is a Scheme FRP system with signals where time can be implicit, as in SuperGlue. By allowing time to be implicit, signals in FrTime and SuperGlue can better interoperate with imperative component implementations; e.g., see [19]. On the other hand, implicit time means having to deal with glitches as described in Section 5. The primary difference between SuperGlue and FrTime is that FrTime is a general-purpose language while SuperGlue is not. Although signals themselves are declarative in FrTime, they are still often manipulated by recursive or higher-order functions with all of the flexibility and complexity that these constructs entail.

General-purpose languages are often extended with connector constructs to improve component-assembly capabilities. ComponentJ [29] and Koala [35] respectively enhance Java and C with procedure-like component connectors. In this area of research, ArchJava [2] is closest in its design goals to SuperGlue. ArchJava is based on enforcing *communication integrity* [22] between components so that they can only communicate through explicitly defined connections. Communication integrity is enforced in ArchJava through static mechanisms, such as alias analysis [3], that allow the controlled use of a general-purpose language across component boundaries. In contrast, communication integrity in SuperGlue is obtained by restricting all inter-component communication through signal connectors.

SuperGlue components, which are bundles of nested objects, resemble in granularity the objects of Simula [6] as well as an actor system [16]. Such objects are not meant as lightweight data structures, as is often the case in Java and C++. They are instead very component-like entities that encapsulated their own state and control and only communicate with other objects through very explicit mechanisms such as message sending. SuperGlue improves on these objects with a simple but expressive language for assembling them together via signals. Finally, SuperGlue's use of nesting resembles the nesting of patterns in BETA [23]. However, as far as we know, our use of inner class constructs as interface artifacts to describe unbounded data structures is unique.

7. CONCLUSIONS

This paper has argued that the complexity of a Turing-complete language is not necessary to express many kinds of component assemblies. Instead, these component assemblies are easier to express in a simple language that can adequately abstract over space and time. In this paper, we introduced SuperGlue as such a simple language, and demonstrated how its support for signals, rules, objects, and arrays can be used to expressively assemble components together.

Even with added expressiveness, simple languages will never replace general-purpose languages: such languages are needed express component implementations. Instead, simple languages can limit general-purpose languages to areas of the program where their power is really needed. Additionally, expressive simple languages, such as SuperGlue, occupy a new language niche between extreme flexibility and extreme ease of use. We believe that many potential programs would benefit from this language niche, where writing such programs with existing languages is currently not practical. Our ultimate goal with SuperGlue is not so much to replace uses of general-purpose languages in existing programs, but rather to make it feasible to write new kinds of programs.

Future work with SuperGlue includes examining how it can be used to build programs in more domains, which mostly involves identifying and building SuperGlue components for these domains. Experimenting with such programs will allow us to reason better about SuperGlue's applicability and performance. Most of our effort is now focused on SuperGlue's third prototype implementation, which enhances SuperGlue with support for live editing, a common visual language feature. With live editing, a SuperGlue program can be changed while it is running, which eases development by enabling "tinkering." The potential for live editing is one more advantage a simple language has over a general-purpose language, where constructs such as recursion and aliasing make it impossible to implement true live editing.

8. 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 ECOOP*, volume 2374 of *Lecture Notes in Computer Science*, pages 334–367. Springer, 2002.

[3] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Proceedings of OOPSLA*, Nov. 2002.

[4] 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.

[5] G. Berry. *The Foundations of Esterel*. MIT Press, 1998.

[6] G. M. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*, 1973.

[7] S. R. Bourne. An introduction to the UNIX shell. In *Bell System Technical Journal*, July 1978.

[8] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: a declarative language for programming synchronous systems. In *Proceedings of POPL*, 1987.

[9] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. To appear in ESOP, 2006.

[10] 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.

[11] E. W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, pages 147–148, 1968.

[12] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of ICFP*, volume 32 (8) of *SIGPLAN Notices*, pages 263–273. ACM, 1997.

[13] A. D. Falkoff and K. E. Iverson. The design of APL. *IBM Journal of Research and Development*, 17(5):324–334, 1973.

[14] D. H. Hansson. *Ruby on Rails*. 37signals, 2005. http://www.rubyonrails.org/.

[15] A. Hejlsberg et al. The LINQ project. http://msdn.microsoft.com/netframework/future/linq/.

[16] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of IJCAI*, 1973.

[17] 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.

[18] IBM. *The Eclipse Project*. http://www.eclipse.org/.

[19] D. Ignatoff, G. H. Cooper, and S. Krishnammurthi. Crossing state lines: Adapting object-oriented frameworks to functional reactive languages. To appear in FLOPS, 2006.

[20] S. C. Johnson. *YACC: Yet Another Compiler-compiler*. Bell Laboratories, 1978.

[21] P.-O. Latour. *Quartz Composer*. Apple Computer, 2005. http://developer.apple.com/graphicsimaging/quartz/-quartzcomposer.html.

[22] D. C. Luckham, J. Vera, and S. Meldal. Three concepts of system architecture. Technical report, Stanford University, 1995.

[23] 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.

[24] Y. Matsumoto. *Ruby: Programmers' Best Friend*. http://www.ruby-lang.org/en/.

[25] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proceedings of OOPSLA*, volume 37 (11) of *SIGPLAN Notices*, pages 211–222. ACM, 2001.

[26] S. McDirmid and W. C. Hsieh. SuperGlue: Component programming with object-oriented signals. To appear in ECOOP, 2006.

[27] OMG. *CORBA/IIOP Specification*, 2.4.1 edition, 2000. Formal document 2000-11-07. http://www.omg.org/-technology/documents/formal/corbaiiop.htm.

[28] D. Rogerson. *Inside COM*. Microsoft Press, 1997.

[29] J. Seco and L. Caires. A basic model of typed components. In *Proc. of ECOOP*, pages 108–128, June 2000.

[30] R. Stallman and R. McGrath. *GNU Make*. GNU. http://www.gnu.org/software/make.

[31] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, Cambridge, MA, USA, 1986.

[32] Sun Microsystems, Inc. *The JavaBeans Components API*. http://java.sun.com/products/javabeans/.

[33] Sun Microsystems, Inc. *The JavaMail API*. http://java.sun.com/products/javamail/.

[34] Sun Microsystems, Inc. *The Swing API*. http://java.sun.com/products/jfc/.

[35] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. In *IEEE Computer*, Mar. 2000.

[36] G. van Rossum and F. L. Drake. *The Python Language Reference Manual*, Sept. 2003. http://www.python.org/doc/current/ref/ref.html.

[37] P. Wadler. The essence of functional programming. In *Proceedings of POPL*, Jan. 1992.

[38] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl*. OŔeilly and Associates, Sept. 1996.

[39] J. Weirich. *Rake–Ruby Make*. http://rake.rubyforge.org/.