Functionally Modeled User Interfaces

Antony Courtney*

Dept. of Computer Science
Yale University
New Haven, CT 06520
antony@apocalypse.org

Abstract. *Fruit* is a new user interface toolkit based on a formal model of user interfaces. This formal basis enables us to write simple, concise *executable specifications* of interactive applications. This paper presents the Fruit model and several example specifications. We consider one example (a basic media controller) in detail, and contrast the executable specification style of Fruit with a more traditional "rapid prototype" implementation using an imperative, object-oriented toolkit (Java/Swing) to show the benefits of our approach.

Keywords: formal methods, executable specifications, user interface toolkits, functional programming, data flow, constraints

1 Introduction

It is widely recognized that programs with Graphical User Interfaces (GUIs) are difficult to design and implement. Myers [14] enumerated several reasons why this is the case, addressing both high-level software engineering issues (such as the need for prototyping and iterative design) and low-level programming problems (such as concurrency). While many of these issues are clearly endemic to GUI development, our own subjective experience (shared by many others) is that even with the help of state-of-the-art toolkits, GUI programming still seems extremely complicated and difficult relative to many other programming tasks.

Historically, many difficult programming problems became easier to address once the theoretical foundations of the problem were understood. To cite just one example, precedence-sensitive parsers became much easier to implement after the development of context-free grammars and the Backus Naur Formalism. In contrast, while some formal models of GUIs have been proposed [7, 3], this work has been largely divorced from the world of practical GUI toolkits. To see this, we need only ask the question "what is a GUI?" in the context of any modern GUI toolkit. In all toolkits that we are aware of, the answer is either entirely

* This material is based upon work supported in part by a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

informal, or depends on artifacts of the toolkit implementation, such as objects, imperative state, non-deterministic concurrency or I/O systems, each of which has an extremely difficult and complicated formal semantics in and of itself.

This situation lead us to pose the following questions:

- While a formal account of GUIs based on objects, imperative programming, and I/O systems is clearly *sufficient*, are such concepts *necessary*?
- Is there a simpler formal model of GUIs that is still powerful enough to account for GUIs in general?

To answer these questions, we have developed *Fruit* (a Functional Reactive User Interface Toolkit) based on a new formal model of GUIs. Fruit's foundational model (called *Yampa*) is based on two simple concepts: *signals*, which are functions from real-valued time to values, and *signal functions*, which are functions from signals to signals. GUIs are defined compositionally using only the Yampa model and simple *mouse*, *keyboard* and *picture* types.

While there are many possible formal models of GUIs, we believe the Fruit model is compelling for a number of reasons:

- The concepts of *signal* and *signal function* in the Fruit model have direct analogues in digital circuit design and signal processing. This allows us to borrow ideas from these established domains, and also resonates with our own experience, in which programmers speak of "wiring up" a GUI to describe writing event handlers.
- Fruit specifications are *extremely* concise. Small interactive GUIs can be written with one or two lines of code, with minimal attention to the kind of boiler-plate that plagues modern GUI toolkits.
- The Fruit model enables a clear account of the connection between the GUI and the non-GUI aspects of the application, and allows a clear separation of these aspects using the Model/View/Controller design pattern [12].
- The Fruit model makes data flow explicit. As we will discuss in detail in section 4, capturing the pattern of data flow relationships explicitly is fundamentally useful when reasoning about implementations of graphical interfaces.

Since we are interested in relating our formal model to real toolkit implementations, we have developed a prototype implementation[1] of the Fruit GUI model as a library for the non-strict functional language Haskell. We chose Haskell as a host language because we can use Haskell's lazy evaluation to design control structures well suited to our model, while leveraging Haskell's base language features (such as expression syntax, functions, type system, etc.). However, we wish to emphasize that the Fruit model is independent of the host programming

[1] We refer to our implementation as a "prototype" because we do not yet provide a complete set of pre-packaged GUI components (widgets) and because the performance of our current implementation is too slow to be considered a production toolkit. However, the prototype is capable of running all of the examples presented in this paper, and many others.

language. Previous work has explored how Fruit's foundational model could be embedded in Java, and we have also explored a simple visual syntax that we use in many of the diagrams in this paper. We will explain Haskell syntax as it is introduced, so that no previous knowledge of Haskell is required to read this paper.

The remainder of this paper is organized as follows. In section 2 we define the Fruit model, and give simple but precise definitions for some useful primitives. In section 3 we present a small example (a control panel for a media player) in detail to demonstrate how Fruit is used to specify basic GUIs. In section 4, we compare the Fruit specification of the media player with a corresponding imperative implementation to clarify the benefits of functional modeling. Section 5 discusses dynamic user interfaces and scalability issues. Section 6 discusses related work. Section 7 presents our conclusions and describes plans for future work.

2 Model

In this section, we present our model of reactive systems and show how GUIs can be accommodated within that model.

Foundations: Yampa

Yampa is a Haskell library for writing declarative, executable specifications of reactive systems. Yampa is based on ideas from Functional Reactive Animation (Fran) [5] and Functional Reactive Programming (FRP) [24], adapted to the *Arrows* framework recently proposed by Hughes [9]. *Yampa* is based on two central concepts: *signals* and *signal functions*.

Signals A *signal* is a function from *time* to a value:

$$\textbf{Signal}\ a \quad = \quad Time \rightarrow a$$

Time is continuous, and is represented as a non-negative real number. The type parameter a specifies the type of values carried by the signal, and is analogous to a *template parameter* in C++. For example, if *Point* is the type of a two-dimensional point, then the time-varying mouse position has type *Signal Point*. As another example, if *Picture* is the type of a single visual image, then *Signal Picture* is the type of a continuous, time-varying Picture, i.e. an animation.

Signal Functions A *signal function* is a function from *Signal* to *Signal*:

$$\textbf{SF}\ a\ b \quad = \quad Signal\ a \ \rightarrow\ Signal\ b$$

We can think of signals and signal functions using a simple circuit analogy, as depicted in figure 1. Line segments (or "wires") represent signals, with arrows

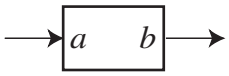

Fig. 1. A Signal Function, **SF** *a b*

indicating the direction of flow. Boxes (or "components") represent signal functions. If we connect the signal function's input port to a *Signal a* value, we can observe a *Signal b* value on the signal function's output port.

A Yampa *specification* defines a signal function. In order to ensure that Yampa specifications are executable in a reasonable amount of time and space, we further require that all signal functions are *causal*: The output of a signal function at time t is uniquely determined by the input signal on the interval $[0, t]$. In order to guarantee this causality requirement, Yampa provides a small number of primitive signal functions and composition operators. All of the primitive signal functions obey causality, and all of the composition operators preserve causality.

Composing Specifications: A Simple Example

Fruit is a modest library of types and functions for specifying graphical user interfaces using Yampa. To illustrate the essence of composing a Fruit specification, consider the following type:

```
type SimpleGUI = SF GUIInput Picture
```

The `GUIInput` type represents an instantaneous snapshot of the keyboard and mouse state (formally just a tuple or record). The `Picture` type denotes a single, static visual image.

A `SimpleGUI`, then, is a signal function that maps a *Signal* of `GUIInput` values to a *Signal* of `Picture` values. As an example of a `SimpleGUI`, consider a challenge for GUI programming languages posed by Myers many years ago [13]: a red circle that follows the mouse. For the moment we assume the Fruit library provides a signal function, `mouseSF`, that can extract the mouse's current position from the `GUIInput` signal:[2]

```
mouseSF :: SF GUIInput Point
```

We will assume the existence of a graphics library that defines basic types and functions for static 2-D images, such as points, shapes, affine transforms and images; see Pan [4] for the formal details. Using this graphics library, we can write: [3]

[2] Haskell-ism: The `::` is a type declaration, and should be read as "has type"

[3] Haskell-ism: In order to support Currying and partial application, Haskell follows the lambda calculus tradition of writing function application as juxtaposition: $f(x, y)$ in traditional mathematical notation is written as `f x y` in Haskell.

```
-- a red ball positioned at the origin:
ball :: Picture
ball = withColor red circle

moveBall :: Point -> Picture
moveBall p = translatePic ball p
```

Given a point `p` whose components are x and y, `moveBall p` is a picture of the red ball spatially translated by amounts given by x and y on the respective axes.

Note that `moveBall` is a function over static values, not over signals. However, we can use Yampa's primitive *lifting* operator to *lift* the `moveBall` function to obtain a signal function that maps a *time-varying* point to a *time-varying* picture (of type `SF Point Picture`). Given any static function `f` of type `a -> b`, `liftSF f` is a *signal function* that maps a *Signal a* to a *Signal b*. Lifting denotes *point-wise* application of the function: If the input signal to the lifted function has some value x at time t, then the output signal has value $f(x)$ at time t.

To allow the mouse to control the ball's position we connect the output signal of `mouseSF` to the input signal of the lifted `moveBall` using serial composition, as shown in figure 2.

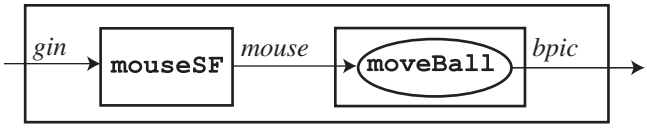

Fig. 2. ballGUI Specification

Although the diagram of figure 2 is a faithful representation of the specification, we use a textual concrete syntax. Our concrete syntax consists of Haskell syntax with some modest syntactic extensions [19] that enable us to directly transliterate data flow diagrams into textual form.[4] Note that, in figure 2 we have explicitly labeled the signals inside of `ballGUI`: `gin` (the `GUIInput` signal to `ballGUI`), `mouse` (the `Point` signal produced by `mouseSF` and fed as input to the lifted form of `moveBall`), etc. In our concrete syntax, we would write `ballGUI` as:

```
ballGUI :: SimpleGUI
ballGUI = proc gin -> do
  gin   >- mouseSF         -> mouse
  mouse >- liftSF moveBall -> bpic
  bpic  >- returnA
```

[4] We have also considered implementing a programming environment that would allow the user to compose specifications directly as data flow diagrams. However, such visual language environments present many substantial UI design and implementation problems in their own right, and would probably still need an internal representation very similar to the linear textual syntax presented here.

The above definition is read as follows: The `proc` keyword introduces a signal function definition. Haskell uses indentation to specify scoping (as in Python), so the body of the `proc` extends all the way to the `returnA` on the last line. Immediately following the `proc` keyword is a *pattern*. This pattern introduces a variable, `gin`, that is matched *point-wise* against the input signal. By *point-wise*, we mean that at *every* sample point, `gin` is bound to a snapshot (sample) of the corresponding signal.

Following the `do` keyword are a sequence of *wiring definitions* of the form:

$$exp \texttt{ >- } sf \texttt{ -> } pat$$

where *exp* is an *expression* specifying the input signal, *sf* is a signal function, and *pat* is a pattern that introduces identifiers bound point-wise to samples of the output signal.

The left-hand side of each wiring definition can be any expression. Each such expression is computed point-wise (i.e. at every sample time). Since the lifting operator `liftSF` denotes point-wise function application, we can often replace explicit use of `liftSF` with a slightly more complex expression on the left hand side of a wiring pattern. For example, we can rewrite `ballGUI` more concisely as:

```
ballGUI :: SimpleGUI
ballGUI = proc gin -> do
  gin            >- mouseSF -> mouse
  moveBall mouse >- returnA
```

Because the expression `moveBall mouse` is computed point-wise, this specifies that, at every point in time, the output signal of the entire `proc` is `moveBall` applied to `mouse`, where `mouse` is the point-wise sample of the output signal of `mouseSF`. While this is just a syntactic shorthand (the data-flow diagram is the same in both versions), this latter version of `ballGUI` helps clarify the connection between point-wise expressions and one-way constraints. We can interpret the last line as a *constraint* specifying that, at every point in time, the output picture produced by `ballGUI` must be `ball` translated by the current `mouse` position.

Discrete Event Sources

While some aspects of user interfaces (such as the mouse position) are naturally modeled as continuous signals, other aspects (such as the mouse button being pressed) are more naturally modeled as *discrete events*. To model discrete events, we introduce the `Event` type:

```
data Event a = EvOcc a -- an event occurrence
             | NoOcc -- a non-occurrence
```

The above is an *algebraic data type* with two alternatives, analogous to (for example) a *discriminated union* type in Modula-2. This declaration specifies that, for any type T, all values of type `Event` T are either of the form `(EvOcc` v `)` where v is a value of type T, or of the form `NoOcc`.

A signal function whose output signal carries values of type `(Event T)` for some type T is called an *event source*. If the event source has an occurrence at time t, then sampling the output signal of the event source at t will yield a value `(EvOcc` v`)`. At non-occurrence times, sampling yields the value `NoOcc`. The value v carried with an event occurrence may be used to carry extra information about the occurrence.

What is a GUI?

The `SimpleGUI` type is sufficient for describing `GUIs` that map a `GUIInput` signal to a `Picture` signal. This accounts for the visual interaction aspects of a GUI, but real GUI-based applications connect the GUI to other parts of the application not directly related to visual interaction. To model these connections we expand the `SimpleGUI` definition to:

```
type GUI a b = SF (GUIInput,a) (Picture,b)
```

The input and output signals of `SimpleGUI` have been widened by pairing each with a type specified by a type parameter. These extra *auxiliary semantic signals* enable the GUI to be connected to the non-GUI part of the application.

Library GUIs

The Fruit library defines a number a number of standard user interface components (or "widgets") found in typical GUI toolkits as `GUI` values. Here we briefly present the programming interface to these components. Note, however, that there is nothing special or primitive about these components; they are just ordinary `GUI` values, defined using the Yampa primitives and graphics library.

Labels The simplest standard GUI components are labels, defined as:[5]

```
flabel :: LabelConf -> GUI LabelConf ()

ltext :: String -> LabelConf
```

A label is a GUI whose picture displays a text string taken from its auxiliary input signal, and produces no semantic output signal.

The behavior and appearance of a component at any point in time is determined by its *configuration options*. `LabelConf` is the type of configuration options specific to the `flabel` component. For labels, `LabelConf` has just one constructor, `ltext`, which specifies the string to display in the label. Note, too, that `flabel` is defined as a function that takes a `LabelConf` argument and returns a `GUI`. The `LabelConf` argument allows the user to specify an *initial default configuration* for the properties of the GUI, analogous to the role of constructor arguments in object-oriented toolkits. If a value for a particular property is specified by time-varying input signal to the GUI, the value specified in the input signal will override the initial configuration.

[5] Haskell's unit type (written `()`) is the type with just one value, also called unit, and also written as `()`. Unit serves a similar role to the `void` type in ANSI C.

Buttons A Fruit button (`fbutton`) is a GUI that implements a standard button control. The declaration of `fbutton` is:

```
fbutton :: ButtonConf -> GUI ButtonConf (Event ())

btext :: String -> ButtonConf
enabled :: Bool -> ButtonConf
```

There are two constructors for the `ButtonConf` type: one to specify the string to display in the button, and another to control whether the button is enabled. A button that is disabled will have a grayed-out appearance, and does not react to mouse or keyboard input. A button is an event source that has an occurrence when the primary mouse button is pressed when the mouse is positioned over the button. Each event occurrence on the output signal carries no information other than the fact of its occurrence, hence the type `Event ()`.

3 A Basic Example

As a concrete example of a Fruit specification, consider the classic VCR-style media controller illustrated in figure 3. The control panel provides a user interface to the simple finite state machine shown in figure 4. Each button is only enabled if pressing the button is a valid action in the application's current state.

Fig. 3. Basic Media Controller

The implementation of the media controller in Fruit is easily derived from the state machine in figure 4, and is shown in figure 5. The implementation uses an enumerated type to encode the current state[6]:

```
data MediaState = Playing | Paused | Stopped
```

Each of the three buttons is made by `fbutton`, and *playE*, *pauseE* and *stopE* are the output signals from each button (of type `Event ()`). Each event occurrence is *tagged* with its corresponding state, and these event signals are *merged* to form a single event signal, *nextStateE*, whose occurrences carry the next state. The *nextStateE* event signal is fed to the `hold` primitive to form a continuous signal, *state*, representing the current state. The `hold` primitive provides

[6] In C, this might be written as:
`typedef enum {PLAYING, PAUSED, STOPPED} MediaState;`

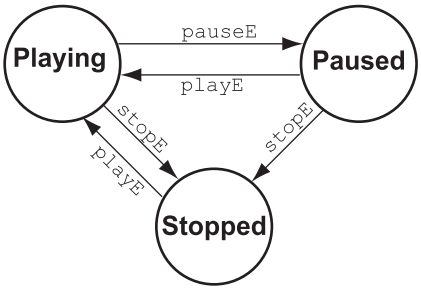

Fig. 4. Media Controller Finite State Machine

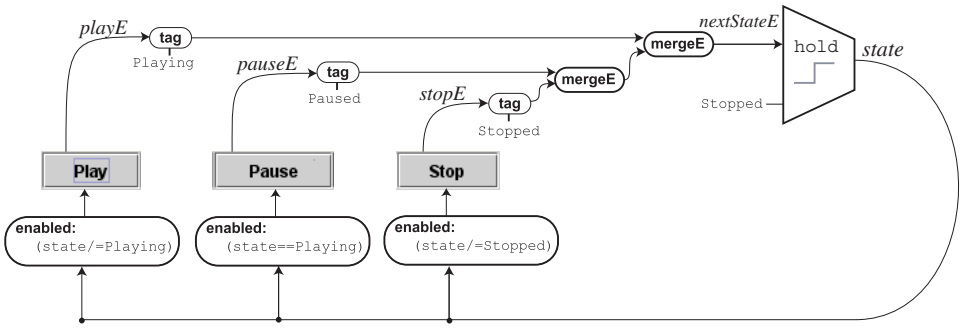

Fig. 5. Media Controller Implementation

a continuous view of a discrete event signal by "latching" (or *hold*ing) the value of the last event occurrence across a period of non-occurrences, as illustrated in figure 6. Finally, the enabled property of each button is determined by a simple predicate applied point-wise to the *state* signal. Each equation is derived directly from the state transition diagram of figure 4 by inspection.

Note that this diagram only illustrates the wiring of the auxiliary semantic signals of each button; the `GUIInput` and `Picture` signals have been omitted. This is because we have abstracted the common pattern of simple horizontal or vertical layout of GUIs into a new type, `Box`, that captures this pattern. A `Box` is a sequence of GUIs that will be arranged horizontally or vertically. `Box` values wire the `GUIInput` and `Picture` signals of each child GUI to obtain a linear arrangement of the `GUIs` in the order in which they appear in the program text, so only the auxiliary semantic signals of each child GUI need to be specified explicitly.

The textual syntax for the media controller corresponds directly to the visual syntax of figure 5:

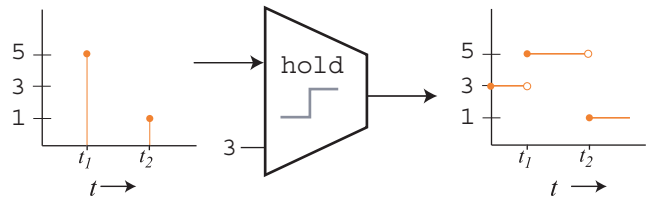

Fig. 6. Semantics of hold

```
playerCtrl :: GUI () MediaState
playerCtrl = hbox ( proc _ -> do
  enabled (state /= Playing)
      >- fbutton (btext "Play") -> playE
  enabled (state == Playing)
      >- fbutton (btext "Pause") -> pauseE
  enabled (state /= Stopped)
      >- fbutton (btext "Stop") -> stopE
  (mergeE (tag playE Playing)
          (mergeE (tag pauseE Paused)
                  (tag stopE Stopped)))
      >- boxSF (dHold Stopped) -> state
  state >- returnA )
```

While the code follows directly from the diagram, a couple of points are worthy of mention:

First, this definition makes use of *recursive* bindings. In this case, `state` is used in the expression for the input signals on the first three lines, but is not defined until the line preceding `... >- returnA`. The formal account of recursive bindings is straightforward, using a standard fixed point operator. As in digital circuit design, there must be an infinitesimal delay somewhere on the feedback path to ensure that the recursion is well-formed. In this example, the `dHold` primitive introduces a delay between its event signal input and continuous output. While the introduction of delays might appear subtle and arbitrary at first glance, in practice it is almost always obvious where to introduce the delays in a specification.

Second, the `boxSF` function lifts an ordinary signal function into `Box`; such lifted signal functions have no visual appearance in the final GUI. The function `hbox` evaluates to a GUI with the contents of its `Box` argument laid out horizontally.

To complete the interface of figure 3, we place `playCtrl` and a label in a vertical box, and connect the *state* output signal of `playCtrl` to the input signal of the label:[7]

[7] Haskell-isms: `show` here has type `(MediaState -> String)`; the `++` operator is string concatenation.

```
player :: GUI () ()
player = vbox ( proc _ -> do
  () >- box playerCtrl -> state
  (ltext ("state: " ++ (show state))) >- label )
```

Once again, the connection between point-wise computations and one-way constraints is apparent in the specifications of `playCtrl` and `player`: We can interpret the input signal to each button as a constraint specifying the relationship between the `enabled` property of the button and a predicate applied to the *state* signal. Similarly, we can interpret the input signal to the `label` as constraint that specifies that, at every point in time, the label's text property must be equal to the given string expression computed from `state`.

4 Evaluation

Fruit provides a formal model of user interfaces, and demonstrates that this model can be used as the basis for a GUI toolkit. But is there any practical benefit to functional modeling? After all, an experienced GUI programmer could implement the media player example in a few minutes using their favorite imperative language and GUI toolkit. At first glance, the specification in figure 5 (or its corresponding textual syntax) may even seem somewhat *more* complicated than a corresponding imperative program, since it involves both an explicit `hold` operator to introduce local state and a feedback loop.

To see why Fruit specifications are useful, consider how the media controller might be implemented in a modern, object-oriented imperative toolkit, such as Java/Swing. A good object-oriented design would encapsulate the current state of the media controller into a *model* class that supports registration of *listener* classes to be notified when the model's state is updated. At initialization time, the implementation would create the model and the button instances, register listeners on the model instance that update the enabled property of the buttons, and register listeners on each button instance that update the state of the model, as illustrated in figure 7. As this diagram illustrates, a feedback loop exists at runtime in this object-oriented imperative implementation, just as it does in the Fruit specification. In fact, a more accurate diagram would repeat this cyclic graph structure once for each of the other two buttons, with each sub-graph sharing the same model instance – a considerably more complex structure than figure 5.

The key difference between figures 5 and 7 is that the former is a diagram of a static specification, while the latter is a visualization of a partial snapshot of the heap at runtime. In the Swing implementation, the feedback loops are hidden from the programmer in the listener lists in the implementation of the model and button classes. Even with whole program analysis, there is no reliable, systematic way for either the programmer or a programming environment to recover figure 7 directly from the source code of the Java/Swing implementation. In contrast, figure 5 is isomorphic to the (static) text of the specification. In short, a Fruit

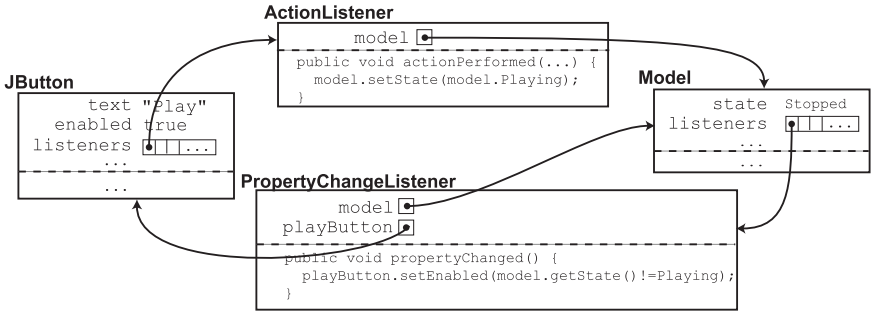

Fig. 7. Runtime Heap in Java/Swing Implementation

specification differs from an imperative implementation by making data flow dependencies explicit in the specification.

So why is it useful to specify data flow dependencies explicitly?

First, explicit dependencies encourage programmers to think in terms of time-invariant *relationships* between different components of the application. The considerable literature on constraints has made the case quite well that this is a higher-level view of user interfaces. Instead of writing event handlers that update mutable objects in response to individual events, the Fruit model encourages writing declarative *equations* that specify the relationships between components of the interface that *must hold* at every point in time.

The data flow style also eliminates a small but important class of programming errors. In traditional imperative event handlers, every event handler must include code to update all of the appropriate objects in response to the event. A common source of subtle bugs in imperative GUI programs is forgetting to update some particular object in response to a particular event, or (even worse) updating the local state of an object, but forgetting to notify registered listeners. In contrast, point-wise dependencies in Fruit are propagated automatically by the implementation.

Making data flow dependencies explicit also enables precise specification of *design patterns* related to data flow. For example, the classic Model / View / Controller (MVC) design pattern [12] enables multiple interactive views of the same underlying data set, and has become the cornerstone of modern object oriented GUI toolkits. The essence of MVC is decoupling of the time-varying application state (the *model*) from the graphical interface, so that the model may be observed and updated by multiple user interface objects. This decoupling can be expressed in Fruit by simply decoupling the state accumulation primitive (`hold`, in the media controller example) from the rest of the GUI. Multiple views and controllers may then be wired to share the same model, and this sharing will be manifest in the specification itself.

Finally, using data flow dependencies as the exclusive mechanism for communication between components of the application enables simple, precise reasoning about causal relationships directly from the specification. For example:

– (forward reasoning): *"What effect does pressing the 'Play' button have on the program state?"* This is easily determined from figure 5 by tracing the path from the play button to the *state* signal.
– (backwards/dependency reasoning): *"What GUI components affect the state?"* This is easily determined by tracing backwards along all signal lines that enter the `hold` primitive that forms the *state* signal.
– (component interactions): *"How does the 'Play' button affect the 'Pause' button?"* This is determined by tracing the directed path from the first component to the second. Note that if the second component is not reachable from the first, then, since a functional specification can have no hidden side effects, the first component has no effect whatsoever on the second component.

In an imperative setting (even an object-oriented one), this kind of reasoning is simply not tractable. Imperative GUI implementations coordinate their activities via *side effects*: One callback writes to a variable or property that is subsequently read by others. Since any callback may directly or indirectly invoke a method or function that updates the global mutable state used by some other callback, there is no practical method for reasoning about or controlling interactions between different parts of the user interface.

5 Dynamic Interfaces

One valid concern about using the media controller example to compare a functional, data flow style of specification with an imperative implementation is that the data flow graph for this example is static: The user interface has a fixed set of GUI components for the lifetime of the application. What about user interfaces where GUI components appear to be added or removed from the interface at runtime? Such interfaces are easily accommodated in an imperative setting by adding or removing components from the interface and updating listener lists at runtime, using (for example) Swing's `Container` interface.

Dynamic interfaces may also be specified in Fruit, by using the *dynamic collections* features of Yampa [15]. As noted in section 2, signals functions (and hence `GUI`s) are first-class values: they may be passed as arguments, stored in data structures, returned as results, etc. Yampa's dynamic collections primitives exploit the first-class nature of signal functions to maintain a *time-varying* collection of signal functions, all of which are executing in parallel. We have succesfully applied Yampa's dynamic collections primitives to build many highly dynamic user interfaces, such as games and simulations. A full discussion of using these primitives for dynamic user interfaces is outside the scope of this paper; suffice it to say that they enable encapsulation of the dynamic aspects of a user interface without sacrificing modularity or reasoning power.

6 Related Work

There have been numerous previous proposals for formal models of graphical user interfaces, such as User Action Notation (UAN) [8] or Paterno's ConcurTaskTrees [17]. The emphasis in most of these formalisms is typically on modeling *user tasks* [18], i.e. a logical description of actions to be performed by the user to achieve certain goals. Such *task models* fit somewhere between requirements specification and design in the classical software engineering process. In contrast, the formalism presented here is focused solely on user interface *implementations*. Task models are typically very high level, focused on the (human) user, and are not directly executable. In contrast, Fruit specifications are comparitively low level, make no direct mention of the user, and *are* directly executable. Many tools have been proposed to assist in deriving executable user interface implementations from high-level task model specifications [22]. Unfortunately, in most cases the precise semantics of such tools and the code that they generate is an implicit property of the tool's implementation. Since Fruit provides a rigorous formal basis for specifying GUI implementation, it would be very interesting to develop a tool to map high-level task models into low-level Fruit specifications.

Another formalism for modeling user interface implementations is Palanque's Petri net based Interactive Cooperative Objects (ICO) [16]. Like Fruit, ICO enables the programmer to give a precise specification of the run-time behavior of a graphical user interface. The core model of ICOs is Petri Nets, a simple formalism with well-understood semantics. ICOs allow Petri net models to be organized into object-oriented interfaces, in which an object's reaction to method invocations is specified by the Petri net. One key difference between Fruit and ICOs is that where ICOs use objects to organize core models into higher level abstractions, Fruit uses a (functional) host language to provide abstraction capabilities and general computation. An important consequence of embedding in a functional language is that Fruit models retain reasoning power and semantic clarity while still being directly executable.

Data flow models and languages date back to the sixties [20, 11]. Lucid [23], Signal [6] and Lustre [2] are examples of synchronous data flow languages oriented towards control of real-time systems. Jacob et al [10] propose a data flow model for user interfaces, including both continuous variables and discrete event handlers. However, their model focuses on modeling "post-WIMP" user interaction, and is cast in an imperative, object-oriented setting. In contrast, the Fruit model demonstrates that the data flow model is applicable even in the classical WIMP setting, and and does not depend on objects or imperative programming. As discussed in section 4, we believe that using data flow as the sole basis for our specifications makes reasoning about specifications much more tractable.

In the realm of user interface toolkits, the closest relatives to Fruit are the FranTk [21] and Fudgets [1] toolkits for Haskell. FranTk is similar to Fruit in the sense that it too uses the Fran reactive programming model to specify the connections between user interface components. However, FranTk uses an imperative model for creating widgets, maintaining program state, and wiring of cyclic connections. The programming interface to Fudgets is very similar to that

of Fruit, although Fudgets is based on discrete, asynchronous *streams*, whereas Fruit is based on continuous, synchronous *signals*. Another key difference is that the Fudgets system allows any Fudget to perform arbitrary I/O actions, whereas such connections to the I/O system would have to be made explicitly in Fruit.

7 Conclusions and Future Work

This paper presented Fruit, a new user interface toolkit based on a synchronous data flow model of reactive systems, and some small example specifications using Fruit. The novel feature of Fruit specifications is that they make data flow dependencies explicit in the specification. Explicit data flow depencies enable simple, precise reasoning about runtime behavior that is difficult or impossible to perform on a traditional imperative, object-oriented program.

We have implemented a prototype of Fruit capable of running all of the examples presented in this paper, and many others, including a small web browser and a "space invaders" video game. Fruit is available for download from `http://www.haskell.org/fruit`.

In the near term, we are interested in developing a highly optimized implementation of Yampa, and in expanding our widget set to include a substantial subset of the components implemented in other modern toolkits. In the longer term, we would like to explore using the visual syntax of figure 5 in an interface builder tool, to enable a designer to specify interface behavior (rather than just static layout) via direct manipulation.

8 Acknowledgements

This work would never have been possible without the contributions of Conal Elliott and Henrik Nilsson, my direct collaborators on Fruit and Yampa. I am also grateful to Paul Hudak, John Peterson and Valery Trifonov for many patient, thoughtful discussions on the ideas presented here. Finally, thanks to Ross Paterson, Magnus Carlsson and many anonymous reviewers for providing valuable feedback on early drafts of this paper.

References

[1] M. Carlsson and T. Hallgren. *Fudgets - Purely Functional Processes with applications to Graphical User Interfaces*. PhD thesis, Chalmers University of Technology, March 1998.

[2] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE : A declarative language for programming synchronous systems. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, New York, NY, 1987. ACM.

[3] A. Dix and C. Runciman. Abstract models of interactive systems. In *Proceedings of the HCI'85 Conference on People and Computers: Designing the Interface*, The Design Process: Models and Notation for Interaction, pages 13–22, 1985.

[4] C. Elliott. Functional images. *(to appear) Journal of Functional Programming (JFP)*, 2001.

[5] C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.

[6] T. Gautier, P. le Guernic, and L. Besnard. SIGNAL: A declarative language for synchronous programming of real-time systems. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 257–277. Springer-Verlag, Berlin, DE, 1987. Lecture Notes in Computer Science 274; Proceedings of Conference held at Portland, OR.

[7] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[8] H. R. Hartson and P. D. Gray. Temporal aspects of tasks in the user action notation. *Human-Computer Interaction*, 7(1):1–45, 1992.

[9] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, (37):67–111, 2000.

[10] R. J. K. Jacob, L. Deligiannidis, and S. Morrison. A software model and specification language for non-WIMP user interfaces. *ACM Transactions on Computer-Human Interaction*, 6(1):1–46, 1999.

[11] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination, queuing. *SIAM J. Applied Math. 14*, (6):1390–1411, Nov. 1966.

[12] G. Krasner and S. Pope. A description of the model-view-controller user interface paradigm in the smalltalk-80 system, 1988.

[13] B. A. Myers, editor. *Languages for Developing User Interfaces*. Jones and Bartlett Publishers, 1992.

[14] B. A. Myers. Why are human-computer interfaces difficult to design and implement? Technical Report CMU-CS-93-183, Computer Science Department, Carnegie-Mellon University, July 1993.

[15] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64, Pittsburgh, Pennsylvania, USA, Oct. 2002. ACM Press.

[16] P. Palanque and R. Bastide. Interactive Cooperative Objects : an Object-Oriented Formalism Based on Petri Nets for User Interface Design. In *IEEE / System Man and Cybernetics 93*, pages 274–285. Elsevier Science Publisher, Oct. 1993.

[17] F. Paterno. *Model-Based Design and Evaluation of Interactive Applications*. Applied Computing. Springer-Verlag, 1999.

[18] F. Paterno. Task models in interactive software systems. In S. K. Chang, editor, *Handbook of Software Engineering & Knowledge Engineering*. World Scientific Publishing Co., 2001.

[19] R. Paterson. A new notation for arrows. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2001)*, September 2001.

[20] C. A. Petri. *Kommunikation mit Automaten.* Bonn: Institut f¨ur Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

[21] M. Sage. Frantk: A declarative gui system for haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, September 2000.

[22] P. A. Szekely, P. N. Sukaviriya, P. Castells, J. Muthukumarasamy, and E. Salcher. Declarative interface models for user interface construction tools: the MASTERMIND approach. In *EHCI*, pages 120–150, 1995.

[23] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language*. Number 22 in A.P.I.C. Studies in Data Processing. Academic Press, London, 1985.

[24] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *Proc. ACM SIGPLAN'00 Conference on Programming Language Design and Implementation (PLDI'00)*, 2000.