A unified framework for control structures in interactive software

Stéphane Chatty

A unified framework for control structures in interactive software

Stéphane Chatty
Université de Toulouse - ENAC
7 av. Edouard Belin, 31055 Toulouse, France
chatty@enac.fr

ABSTRACT
Control structures such as event passing, state machines and data flows help programmers express the behavior of interactive software. But used alone they cannot describe systems in their totality, and when combined with standard control structures their semantics becomes unclear. This article proposes a set of requirements for unifiying all control structures in a general framework. It then proposes a candidate framework in which all software components can be described by processes and their interactions by process couplings. Couplings are the basic block from which control structure such as state machines, data flow connectors, sequences and functions can be derived. We show how a wide variety of programming situations and architecture patterns can be described by combining these control structures. The power of expression and interoperability provided unlock possibilities such as interaction-oriented programming languages.

ACM Classification Keywords: H5.2 Information Interfaces and presentation: User Interfaces; D3.3 Programming Languages: Language Constructs and Features.

General terms: design; human factors; languages

Author keywords: interactive software, programming language, control structure, data flow, state machine, process

INTRODUCTION

Interactive software is long known as costly to produce [26]. This is harmful to high complexity sectors such as aeronautics, where the need for multiplying user interface prototypes to support validation processes comes up against the cost of creating prototypes. This also impacts end users, because programming their digital environment often involves interaction-related concepts [8].

Various fields, from computation to distributed systems and artificial intelligence, have produced general purpose programming languages. Imperative, functional, object-oriented, reactive or logic-based, these languages have evolved until communities of programmers have considered them well founded and adequate for their goals. Interactive software, with its focus on states and events, seems singular enough to justify the emergence of interaction-oriented programming languages. However, most interactive software still relies on a mix of mainstream languages and user interface toolkits that provide additional concepts, with little theoretical foundation.

This article is a contribution toward the emergence of general-purpose interaction-oriented programming languages, in the hope that well defined semantics and wide expressivity will facilitate interactive software development. We focus here on execution control, and on the goal of unifying control structures in a single framework. We first contrast the status of standard and interaction-oriented control structures, and compare their evolutions. We formulate requirements for a control framework, that is a set of control structures that covers the needs of interactive software. We show how, using a process model, known control structures ones can be derived from a single primitive. Relying on an implementation of this control framework and its use in full-size applications, we follow Dix's recommendations for works on theories in HCI [9] and demonstrate through examples how the framework meets the requirements. We conclude on the practical consequences for tools aimed at designers and programmers, including dedicated programming languages.

LIMITS OF HYBRID MODELS

User interface toolkits come with control structures, which are gaining importance as post-WIMP interaction requires designers and programmers to create interactors themselves. Formulating the internal behavior of a button is easier with a state machine than with function calls [1], formulating how a window can be moved and zoomed at the same time is easier with data flows, and so on.

The role of control structures such as events, state machines and data flow has been well established by practice, but it has not replaced the use of traditional control structures. Programmers use mainstream languages such as Java or C++ not only to write the non-interactive part of their software, but also as a host infrastructure to combine control structures.

These hybrid solutions that combine models of execution have practical merits. Still, there would be benefits in having more integrated solutions that clarify the links between models and dispense from using host languages. One of the limits of hybrid models is that they are not self-sufficient and do not provide answers to questions such as:

- data flows and state machines both involve notification; what is the underlying mechanism that explains this? For example, why can pointer movements be used as sources

2015 VERSION
FILED IN 2018 AS hal-01800741

to both data flows (to drive a cursor) and state machines (to implement drag and drop)?

- an on/off switch can be described with a state machine that emits events, but it also has a value (on or off) that could feed a data flow. Why and how can this be used?
- an animation trajectory has similarities with both a computation function and an iterator. What exactly is common, and can it be factorized?

In other contexts programmers are used to well founded models that give clear answers to such questions, and do so because they rely on a few independent primitives from which all other concepts are derived. A Smalltalk programmer who wonders about the similarities between expressions `[3 + 5]` and `[myrect draw]` can use the *object* and *message* concepts to understand the roles of `3` and `+`. A Lisp programmer can understand the similarities between `(mapcar)` and `(reduce)` by returning to *functions* and argument lists.

The benefits of such models go beyond mere comprehension, because they show how the constructs are interoperable, either to connect them or to replace one with another with minimal refactoring. For instance, understanding the links between `(mapcar)` and `(reduce)` allows to reuse functions and lists when modifying an algorithm. At the opposite:

- converting a pointer from event style to data flow style requires code written in the host language (eg. see [2]);
- same for turning an on/off switch into a data flow brick;
- the similarity between animation, iteration and computation cannot be exploited easily in a sequential language.

By showing how standard constructs are made, these models also suggest other combinations and stimulate creativity. For instance, Python *generators* exploit the common roots of iterators and functions in the object-oriented model. Finally, being self-sufficient these models support the use of pivot formats for code analysis, MDE techniques, formal techniques, and graphical programming tools, all of which are partly defeated by the use of host languages in hybrid solutions.

COMPUTATION, INTERACTION: REPEATING HISTORY

Unified models are desirable, but available only when no interaction is involved. Comparing computation and interaction can provide lessons as to how unification can be attained.

Tracing the border between interaction and computation

The split between functional core (FC) and user interface (UI) proposed in the literature does not capture these situations: programmers more and more use hybrid solutions for both FC and UI. In addition, some FCs are interactive software, for instance a plant supervision system. And some interactive programs have no FC or UI worth mentioning, for instance one that says "stop the heating and email me when the temperature is above 20°C".

Consequently, we define interaction-oriented and computation-oriented software using the following distinction: control is internal in computation, external or mixed in interaction [18]. Given this distinction, we can study the evolution of control structures in computation so as to establish requirements and research directions for interaction.

Control structures: from expressiveness to simplicity

Imperative and functional programming have been the dominant paradigms for a few decades. An imperative program is a series of instructions whose order describes their execution sequence. Over time, new control structures have been introduced so as to make algorithms easier to write and to provide simple support for more complex situations: loops, conditions, functions, coroutines, generators, aspects, etc. These extensions were often made at the expense of generality [12] and simplicity: it was expressiveness first.

In contrast, functional programming was advocated for its conceptual simplicity [3]. A program is a function that calls other functions, and control structures are provided as functions. Functions are the root primitive of the control framework, which gives strong internal consistency. Later, the introduction of object-oriented programming and prototype-oriented programming brought a similar measure of conceptual unification to imperative programming, all instructions being considered as message exchanged with objects [28].

This history of control structures can be analyzed as the progressive capture of the requirements of a population of users. First starts a process of identifying concerns and embedding them in a design. Then because conceptual simplicity is an important design feature [4, pp. 42-44], starts a second process of unification toward simplicity. This could have ended there, but then programs became interactive and the two design processes started again.

Interaction: new concerns, new control structures

At first, interaction was through terminals and this only required marginal changes, because the query-response style is compatible with standard control structures. Real changes came with graphical user interfaces. Drawing can still be described as sequences of instructions, but the execution of programs became more controlled by external input than by their own sequence of instructions. The concept of external control appeared [15] and events became a new control structure, implemented with extensions such as callback functions or signal/slot patterns.

Then graphics became more and more encapsulated and algorithms disappeared from the concerns of programmers. The behavior and the reuse of interactive components became more central. Programmers started to face the spaghetti of callbacks [23], and new control structures were needed.

Because these new concerns were specific to user interfaces, they were not treated in programming languages but in graphics libraries. State management was one of the earliest, first to describe the global dialogue between the user and the program [17], then to describe the behaviors of individual components [24]. Attempts were also made at managing state in combination with another growing concern: the need to organize the complexity of programs, using hierarchical systems [14]. Other concerns appeared when the interaction

styles became more continuous: animation, drag and drop, data visualization. Various forms of data flow control were proposed to support the description of data representation and graphics layout [25], animation [5], and input [5, 10].

Interaction: seeking simplicity again

What interaction programmers have gained during this process for their specific needs, they have lost in simplicity. Various attempts have been made at getting consistency back.

Taking advantage of similarities between data flow and the flow of values in function calls, functional reactive programming combines some features of reactive programming with the syntax of functional languages [11]. Similarly, some have used the extensibility of the syntaxes of object-oriented languages to integrate the new control structures (mainly state machines) as much as possible [19, 1], obtaining some form of syntactical consistency. The C# programming language also provides the same consistency for events.

Conversely, it has been proposed to structure programs using Petri nets and to use an object-oriented language for making computations when the transitions are fired [27]. Similarly, several authors have proposed to combine data flows and state machines by letting the state machines control which data flows are active at a given time [5, 16, 7].

All these are partial solutions, that lead to the situation described earlier in this article. A more complete integration was described in [6], where events and state machines are unified and data can be used as event sources and as data flow feeds, but the unification of states and values was incomplete and imperative programming was still required for some parts of programs. Full unification is still an open challenge, addressed in the rest of this paper.

REQUIREMENTS FOR CONTROL FRAMEWORKS

The above analysis allows to define a set of requirements for the design of a set of control structures, aimed at future languages for interactive software or at future tools for designers. All these requirements are met by functional languages and object-oriented languages for computation-oriented software, but not for interaction-oriented software.

Conceptual unification

The ability to place all control structures with regard to others or, even better, to understand how they all derive from a single primitive, is a major goal. It brings semantical interoperability and flexibility. It also brings conceptual integrity and simplicity, which are highly valued not only in computer science, but also in most science and design disciplines (eg. unification in physics). A number of other requirements depend on this one.

Additionally, the more the concepts are consistent with those used by programmers or designers in their daily life, the better. This was sought in the design of object-oriented languages, with explict references to philosophical concepts. For interactive software, this goal is challenging because it encompasses notions such as physical objects and human activities or procedures.

Coverage

The set of control structures must cover all the concerns usually met by programmers and designers, whatever the interaction style they choose. Sequences, functions, loops, and tests are needed for command line dialogs and for computation. State machines are used for widgets, direct manipulation and dialog design. Data flows are useful for visualization, animation and direct manipulation. Parallelism and synchronization primitives are required by multimodal interaction. Aspect-like control structures are useful for adaptive user interfaces [21]. In addition, new domains such as the Internet of things will probably bring their own requirements.

Extensibility

Since new interaction styles bring new concerns, and possibly new control structures, the framework must be extensible. Given the diversity of possible interactions styles, programmers should be able to create control structures themselves with no need to use another language. This stresses the importance of organizing the set of control structures as a framework in which primitives can be combined at will.

Compatibility with development processes

The development processes of interactive software require cooperation between different disciplines; for instance a designer may need to modify a behavior written by a programmer. They involve concurrent engineering: designers can produce graphics and behaviors while programmers are writing algorithms, on the basis of a predefined application architecture. They also are iterative, with successive refinements and extensions of the same design.

This variety of actors and tools (programming languages, graphical editors, transformation tools) pleads in favor of a further unification, available in languages such as LISP: common concepts for code and data, which favor the exchange of code between tools.

Support for formal analysis

A number of stakeholders are in demand of better support for analyzing interactive software and its behavior: safety experts who need to validate the software against a specification, programmers who want to check their code against various behavior requirements, human factors professionals who want to predict the performance of end-users with the software, etc. The closer the control structures are to a form that can be processed automatically, the better.

SELECTING AN EXECUTION MODEL

The rest of this article describes a control framework that meets the above requirement. We start by analyzing two design decisions on which the proposed solution relies: the use of the reactive programming paradigm, and an execution model that supports multiple scale analysis.

Formally describing the chosen execution model is beyond the goal of this article. Instead we highlight the reasons for our design decisions and outline the basic concepts of the execution model, in order to root the control framework in these concepts and to help the reader assess how the requirement of *support for formal analysis* can be met.

The role of the execution model

When using two sets of control structures, one for computation and one for interaction, it is sometimes difficult to understand how programs are executed. For instance, in the code below the two animation sequences and the message printing do *not* occur in the order that the inattentive reader would think: the `done` message is printed first.

```
void sequence () {
  a1 = Animate (rect, 100, 100);
  a2 = AnimateAfter (a1, rect, 50, 50)
  printf ("done");
}
```

The reason is that the two `Animate` calls do not execute the whole animation sequences: they merely start them and return. There is no standard way in any sequential language to express ordering constraints between animation sequences.

Most control structures have a clear meaning in a given execution model only: loops and functions are derived from sequential programming, Prolog's cuts are meaningful only within its backtracking system, etc. The control structures used for interaction, such as state machines, events and data flows, are generally associated with reactive models, e.g. in Lustre [7]. This is consistent with the concurrency exhibited by user interface software: graphical components have concurrent behaviors with very few ordering constraints, animations need to execute in parallel, input and output devices have an intrinsically concurrent behavior, and so on.

Relying on the reactive paradigm for interactive software seems a natural choice to avoid the confusions described above. Not only does it provide the appropriate basis for interactive control structures, but since models of concurrency are supersets of sequential execution models [22] it also provides appropriate semantics for classical control structures.

Choosing an execution model also has consequences on the *development processes* requirement. When working on a component, programmers and designers frequently need to shift from one level of granularity to another. For instance, designers sometimes improve graphical buttons by replacing their atomic state changes with more complex animation sequences. Similarly, an atomic value assignment can be replaced with a dialog that asks confirmation of the change. Therefore, it is desirable that the execution model allows to change the level of analysis without difficulty, like functional programming makes its easy to replace a literal expression with a function call.

A hierarchical process model

The model chosen here relies on a hierarchy of interacting processes, derived from common process algebras in which a process is an abstraction defined by the signals it exchanges with other processes [20].

Everything is a process

All that happens on the computer can be modeled by a **process**. This includes the behavior of interactive components such as buttons and menus. This includes components themselves, made of all their internal processes. This includes animation as well. But this also includes many other things: entities as diverse as graphical objects, physical input devices, and memory cells can all be modeled as processes if need be. This helps meeting the *conceptual unification* requirement in its more ambitious version, because it covers both the software and its environment.

Hierarchy and couplings

Each process can be considered as made of other processes. This allows to describe all systems of interest at the chosen level of granularity, whether for hardware or software. For instance, graphical objects can be considered as atomic processes for some purposes. Alternatively, they can be modeled as a combination of memory cells and rendering routines.

Some processes, such as the contact between two electrical conductors or the interconnection between two software components, belong to two or more parent processes. We call them **couplings**. Some couplings are predefined, such as the coupling between memory cells and rendering routines in a graphical object or the physical connections in an input device. Others are created by programmers. Dor instance a programming instruction that modifies a variable creates a coupling between a program and the computer memory.

Activation and interaction

The evolution of a system is described by the **activation** of its sub-processes. Activating a lamp makes it emit light, activating an assignment instruction modifies memory, activating a dialog box starts an interaction sequence.

When two processes are coupled, activating one can trigger the activation of the other through that of the coupling. This is called an **interaction**. Because processes are hierarchical, interactions are hierarchical too: a gesture on a touch screen is an interaction between the finger and the sensors of the touch screen, made of a series of shorter interactions. The smallest interactions between software components are named **events.**

In a computer, the hardware environment and the operating system already implement couplings. For instance, physical couplings in a mouse are designed so that movements of the hands trigger electronic sensors, and these sensors are coupled to software components that represent the mouse in the operating system. Couplings created by application programmers enrich the wealth of couplings that already constitute the user's environment. They define how programs become part of this interactive environment, and how the components inside the programs interact together.

AN INTERACTION-ORIENTED CONTROL FRAMEWORK

The proposed control framework relies on the above hierarchical process model. It is based on a single primitive that can be derived into all the required control structures by combining it with the basic operations on memory. We describe below how control structures from the literature are defined and demonstrate their use, thus showing how the *coverage* requirement is met. We also give an example of how new control structures can be created when required, so as to address the *extensibility* requirement.

All the examples in the rest of this article have been implemented using a component-based programming environment named djnn, available at `http://djnn.net`. djnn has three

goals: supporting innovative interactive software, supporting modern development processes, and paving the way to interaction-oriented programming languages. djnn is available as a toolkit with APIs in C, C++, Java, Python and Perl. Alternatively, it can be used as an XML interpreter so as to implement components in a portable format. The contents of these XML files are hierarchies of components, and can be understood as abstract syntax trees resulting from the compilation of more usable notations.

In order to save space, the examples are formulated in compact pseudocode rather than in XML format or with any djnn API. For instance, we write:

```
component c {
  rectangle r (0, 0, 10, 10);
}
```

instead of the following XML code:

```
<djnn:component id="c">
 <svg:rect id="r" x="0" y="0" width="10" height="10"/>
</djnn:component>
```

Control structures as components

A component-oriented environment can be defined from the execution model defined above by considering that each component is the implementation of a given process. Programming then consists in creating components and couplings between them. Coupling components makes new interactions possible between them, thus defining the behavior of a program. For instance, by coupling the position of the mouse to a graphical object, one creates a cursor that is updated every time the mouse moves. Note that this coupling has the effect of creating an indirect coupling between the user and the graphical object, thus allowing the user to interact with the cursor. Similarly, one can create an animation sequence by coupling the completion of an animated effect to the start of another. By coupling operations to the start of an application, one can also create a traditional program that runs computations as soon as it starts.

All control structures are aimed at creating couplings, each control structure being dedicated to a pattern of coupling. In order to meet the *development processes* requirement as much as possible, control structures are defined as components themselves. This allows to manage them like any other components, thus contributing to requirement *unification* and allowing programmers to create programs by creating and assembling components. We will also see later that it contributes to *unification* and *extensibility* by allowing to create couplings between control structures.

Control primitive: the binding

The most elementary programming instruction consists of creating a single coupling: *"when component A is activated, I want component B to be activated too"*. Component A is the trigger of the coupling and component B is its action. Trigger and action are just roles that any component can play. For instance, given an integer variable, one can decide to modify this variable as the action of a coupling, just as one can use modifications of the variable as the trigger of a coupling, making it an active value.

The component that manages the creation of a simple coupling is named **binding**[1]. The following pseudocode gives various examples of bindings. It illustrates the variety of entities that can be considered as components: graphical objects, input devices, programmer-defined components, functions, and even components from other programs, resources from the operating system and from other computers. It also stresses the importance of having a complete and consistent system for addressing components. The naming system in djnn is not described in this article, but a hierarchical system similar to URIs can be used in most cases and actions can access their execution context, such as their trigger's properties, through the same naming system.

```
# beeping repeatedly
binding (myclock, beep);
# beeping when an numerical value changes
binding (mygame/score, beep);
# beeping when another application quits
binding (system://application1/quit, beep);
# launching a program when a file is modified
binding (file://eics.tex, system://XLaTeX/reload);
# controlling an animation with a mouse button
binding (mouse/left/press, animation1/start);
binding (mouse/left/release, animation1/stop);
# displaying a help box when an 'h' gesture is made
binding (gestures/h, myapplication/helpbox);
# quitting the application upon a button press
binding (quitbutton/hit, application/quit);
# implementing part of the behavior of a button
binding (quitbutton/rect1/enter, quitbutton/hovercolor);
# chaining two animation sequences
binding (animation1/end, animation2/start);
```

Bindings are not only a very simple control structure. We will use them below to derive all other control structures, ensuring that no reference to the sequential execution model is ever required to explain the reactive behavior of programs. This makes them the root primitive of the control framework.

Since bindings are components, programmers can choose to activate them only at given times, thus creating conditional behaviors. For instance, in the pseudocode below that alternatively emits the sounds `tick` and `tock`, the activation status of bindings `b1` and `b2` is used to create a two-state system.

```
1 binding b1 (myclock, tick);
2 binding b2 (myclock, tock);
3 binding (b1/run, b2/stop);
4 binding (b2/run, b1/stop);
5 binding (b1/trigger, b2/run);
6 binding (b2/trigger, b1/run);
7 b1/run;
```

The two states are implicit in this code. Initially `b1` is active (line 7), and `b2` is stopped (line 3). When the clock goes off, `b1` is triggered; this activates `tick` (line 1) and `b2` (line 5), and stops `b1` (line 4), thus entering a second state. When the clock goes off again, lines 2, 6 and 1 ensure that `tock` is activated and the system returns to the initial state. Note how the hierarchical system is used when referring to the `b1/run`, `b1/stop` and `b1/trigger` subcomponents; here, the `trigger` subcomponent of `b1` refers to the process of detecting that the trigger of `b1` is activated.

[1] "binding" is closer here to the "bind" instruction in Tcl than to data bindings in XAML

State machines

State machine components are a more compact way of obtaining the same result as above. They are useful to describe the behavior of components with a reasonable number of states, such as buttons, menus, many existing post-WIMP components, and probably many that remain to be invented. For instance, the Pinch-Rotate-Zoom behavior on tablets is easily expressed with a state machine.

State machines can be built by combining bindings with components that are able to maintain a state. Being reducible to the same concept of coupling, they can be interchanged and combined at will with bindings. They just make it easier to describe collections of conditional couplings.

Finite state machine components (**FSM**) are composite components that contain other components named **states** and **transitions**. Transitions are special cases from bindings: they are defined with a trigger, and create the same type of coupling as a binding. What makes them special is that 1) each transition is defined relative to a state, named its origin, and is active only when its origin is active, and 2) the action of a transition is always a second state, named its destination. As a consequence, the triggers of the transitions constitute the inputs of the state machine: the machine changes state depending on the sequence of activation of triggers, and ignores events that do not match the current state.

As an example, the code below describes the internal behavior of a software button designed for use with a mouse:

```
component mybutton {
  rectangle r (0, 0, 100, 50);
  fsm f {
    state idle, pressed, out;
    transition press (idle, r/press, pressed);
    transition click (pressed, r/release, idle);
    transition leave (pressed, r/leave, out);
    transition enter (out, r/enter, pressed);
  }
}
```

Because states and transitions are components, they can be used as triggers in bindings and transitions from other FSMs. For instance, when properly renamed to be accessible as `mybutton/click`, the above `click` transition can be used to bind actions to the triggering of the button:

```
binding (mybutton/hit, application/quit);
```

This is the behavior of a Mealy machine. Alternatively, by binding actions to states rather than transitions, one can produce a state machine whose outputs depend only on the states that have been reached and not on the transitions (Moore machine). By combining the two systems, one can for instance produce the visual feedback for the behavior of a widget as well as the appropriate events to trigger other components. We will later see a switch component designed to make this easier.

Combining FSMs by coupling their transitions, or by controlling the activation of one by a state or a transition of another, makes it possible to create complex behaviors without having to introduce more complex structures such as hierarchical state machines. It also makes it easier to structure applications as collections of reusable components.

Properties: data primitives and control structures

The concept of memory is toned down in functional languages in favor of arguments and return values, to avoid so-called side effects. But in interactive software, it is important to explictly represent properties such as the label of a button. In addition, properties also play a role in control flows because their changes are events of interest.

We propose two equivalent definitions of **properties**. They can be defined as state machines with a large number of predefined states (2^{64} states for an integer), and with no support for accessing individual transitions. The value of a property is its state. Properties can also be defined as primitive components that represent the physical memory of a computer, state transitions being implemented by the hardware like the rendering of a graphical object. This duality ensures both a proper foundation for a data model and consistency with the control framework, as illustrated below with data flow.

Data flow

In contrast to state machines, data flow is a programming style for when the flow of propagation of states is more important for programmers than the individual state changes. This often occurs when there are many states or very frequent changes, usually because the various states are just perceived as the result of sampling a value that changes continuously.

Since "value" and "state" are two aspects of the same thing, data flow can be interpreted as a special case of coupling state machines together. With state machines, the default principle is that every transition has a different action. With data flow, all transitions trigger the same action: propagating the new state, that is the new value, to other components. When the user's hand moves on a touch screen for instance, programmers are more interested in deciding what components will receive the flow of values and how they will route and filter it than by handling each event individually. We define two control structures to support such continuous behaviors.

Connecting components

Connector components allow the value of a component to propagate to another component whenever it changes. For instance, the two connectors below ensure that rectangle `rect1` will move with the mouse.

```
connector (mouse/position/x, rect1/position/x);
connector (mouse/position/y, rect1/position/y);
```

A connector is equivalent to a binding with a predefined action that copies the value from its source to its destination. This means that the first line in the above pseudocode is equivalent to the combination of two components below, in which we artificially introduce an "assignment" component to represent traditional assignment instructions.

```
# an assignment that sets rect1/position/x
assignment c (rect1/position/x, mouse/position/x);
# the assignment is bound to changes in mouse/position/x
binding (mouse/position/x, c);
```

There are strong similarities between data flow and functional programming. Functional reactive programming can be interpreted as a syntax that disguises connectors as function calls, and functional programming per se can be seen as a chain of

connectors that is only triggered once when the program is launched. To illustrate this, consider the following data flow chain to the functional expression that follows it. Both represent a simplified version of the chain of computations that transforms the actual data emitted by a physical mouse into what application programmers are used to: the relative movements are "accelerated", then compounded, then cropped to stay within the display area.

```
multiplier mult (3);
adder add (512);
maximum max (1024)
# compute (max 1024 (+ 512 (* 3 physmouse_position_x)))
connector (physmouse/position/x, mult/input2);
connector (mult/result, add/input2);
connector (add/result, max/input);
connector (max/result, softmouse/position/x);
```

Synchronizing flows

The above example is a chain of connected components, used as data filters. Predefined filters can be used, but programmers often need to create their own, like for instance the multiplier component from above. This can be done by putting three numerical components named `input1`, `input2` and `output` as well as a multiplication operation into a larger component, and adding two bindings on the two inputs. Unfortunately, this would have an undesired side-effect: when the two inputs are modified as the result of a single event, the output would be modified twice for that single event and this would lead to inconsistent behaviors. For instance, say that a beep is emitted every time the output changes, and that the inputs are computed from a key press; then there would be two beeps for each key press, where the user expects one.

For this, we introduce a second data flow-oriented structure: the **watcher**. A watcher is a binding with several triggers. It uses synchronization techniques similar to those of reactive languages [7] to ensure that its action is activated only once per asynchronous event, that is once for each event that has external causes. For instance, given a passive component named `multiplication`, the internals of the `multiplier` component would be as follows.

```
component multiplier {
  integer input1;
  integer input2;
  integer output;
  multiplication m (input1, input2, output);
  watcher (input1, input2, m);
}
```

Composite flows

Finally, while we have only used data flow in conjunction with numerical components so far, this applies to other components as well. As already mentioned, the value of a numerical component is its current state. This definition extends to all components that have a state, including those that contain components that have a state. Therefore, connectors can be used between any pair of compatible components to synchronize their state. For instance, the mouse-rectangle connection from earlier can be written:

```
connector (mouse/position, rect1/position);
```

Combining and extending

The components described so far, all built around the same basic blocks, provide unification of the most common control structures used for user interface programming: events, state machines and data flows. Combined together, they can also be used to create new control structures that extend the control framework.

To begin with, components named switches can be created to control what parts of a component are active at a given time. For instance we can enrich component `mybutton` from page 6 with one color for each state:

```
component mybutton {
  ...
  switch s {
    color idle (black);
    color pressed (white);
    color out (grey);
  }
  connector (f/state, s/state);
}
```

A switch is an open collection of components, named its branches; the above switch has three branches. In addition, switches have a property named `state` and a watcher that activates the branch named in this property. When the `state` property is connected to the state of a compatible FSM, this ensures that the FSM controls which sub-component of the switch is active. Here, this makes the button changes color when it changes state.

Switches can be used to convert data flows to states when needed. For instance, in the pseudocode below the color of rectangle changes with the values of a number:

```
integer i;
division d (0, 3);
connector (i, d/input1);
switch s {
  color 0 (red);
  color 1 (green);
  color 2 (blue);
}
connector (d/remainder, s/state);
rectangle r;
```

This also allows for the creation of simple conditions. For instance, the following pseudocode emits a beep when the `x` coordinate of a pointer has a given value, which can be useful for debugging purposes.

```
switch s {
  beep 42;
}
connector (pointer/position/x, s/state)
```

Other classical control structures can be obtained by combining the basic structures described above. For instance:

- sequences can be defined with bindings to the end of components that terminate on their own;
- a graphical scene graph can be considered as a control structure: it tells what graphical objects must be rendered and in which order. This can be defined by managing two sets of bindings: parent-child bindings that ensure that child objects are active only when their parents are active, and bindings that create a sequence between the activation of children so as to control their rendering order;

- a function call is a particular kind of interaction similar to a dialogue fragment: the caller activates the function with some context, the function computes a result, passes it to the caller, terminates, then the caller resumes its own activity. To reproduce this, we define functions as components that terminate their activation on their own. In an imperative setting, calling a function then consists of binding the activation of a function to the termination of the previous instruction in the sequence, and binding the next instruction to the termination of the function. In a functional setting, each function consists of series of function calls: first those needed to evaluate a series of arguments, then the main call performed with these arguments. This structure can be modeled as a simple sequence or a four-state FSM: start, activation of the functions used to compute the arguments, activation of the main function, then termination and return to start.

APPLICATION TO ARCHITECTURE SCENARIOS

Deriving control structures could be used to validate the proposed framework by establishing its Turing-completeness. However, the value of a theoretical result can also be established by selecting examples that demonstrate its applicability [9]. For this reason, we now describe a few scenarios commonly encountered in interactive software programming, so as to illustrate how the control framework works in practical situations and to demonstrate its relevance.

Common architecture patterns

Creating a graphical application consists mostly of assembling components, connecting them, and organizing them to make their structure and their reuse more manageable. djnn supports this by encouraging programmers to create a hierarchy of components, each made of previously existing components. We describe below how a few common situations can be managed with the control framework described above.

Building and initializing components

We have already described how to build a component by assembling other components: `mybutton` is made of four components, a rectangle, a FSM, a switch and a connector. This button is simplistic, but improving it would just require elaborating on the number of states and the number of graphical components. To further illustrate how components can be combined recursively, the pseudocode below describes a dialog box made of two instances of `mybutton`.

```
component mydialogue {
  rectangle frame (0, 0, 300, 100);
  mybutton o ('ok', 30, 25);
  mybutton c ('cancel', 170, 25);
}
```

This dialog box is made of a rectangle and two instances of `mybutton`. For this to work, two changes have to be made to `mybutton`: adding a text, and allowing the text and the rectangle of the button to be initialized with parameters.

The latter point underlines a situation in which programmers currently fall back to imperative or functional programming: initialization. Because in the case of initialization the flow of control is internal like in traditional computation scenarios, function calls are indeed appropriate and very few authors have felt the need to propose alternative solutions. Here, the initialization flow is derived from the relationship between a component and its sub-components: the activation hierarchy is managed by implicit bindings between the parent and each of is children. In this regard, the empty components that programmers can fill to build their own components can be considered as a specialized control structure.

In the new version of `mybutton` below, we represent the sharing of information between parent and children during initialization. We use a syntax similar to arguments in function calls. Note that this is just a syntactic convention: `mybutton` and `rectangle` are not functions, if only because they do not terminate spontaneously.

```
component mybutton (t, x, y) {
  text (t, x, y);
  rectangle r (x, y, 100, 50);
  ...
```

Reusing a component, event style

Programmer-defined components can be used as event sources, like basic components. For instance, the following additions to `mydialogue` show how one ensures that the dialog box disappears when either of its buttons is pressed.

```
component mydialogue {
  ...
  binding (o/hit, stop)
  binding (c/hit, stop)
}
```

Adapting components

Our dialog box can also be used as an event source, but programmers who reuse it expect `ok` and `cancel` events, not `o/hit` and `c/hit`. Such interface adapatations are one of the reasons why patterns such as the Functional Core Adapter and Presentation-Abstraction-Control had to be introduced [13]. The following additions to the dialog box illustrate how the external interface of a component can be adapted using empty components and bindings. More complex adaptations can also be performed, relying for instance on their own state machine to translate events.

```
component mydialogue {
  ...
  component ok;
  component cancel;
  binding (o/hit, ok);
  binding (c/hit, cancel);
}
binding (mydialogue/ok, beep);
```

Communicating state machines

Describing the behavior of applications with Petri nets or extended state machines that communicate among themselves has often been proposed. This is what happens here when two components with state machines are bound, for instance when a button uses the `press` and `release` events that come from the state machine of an input device. We show below how this can occur inside components so as to express complex behaviors; for instance, the following pseudocode simulates a three-state lamp switch controlled by a push button.

```
component myswitch (source) {
  fsm pushbutton {
    state pressed, released;
    transition press (pressed, source, released);
    transition release (released, source, pressed);
  }
  fsm behavior {
    state off, 1, 2;
    transition (off, pushbutton/press, 1);
    transition (1, pushbutton/press, 2);
    transition (2, pushbutton/press, off);
  }
}
```

Reusing a component, data flow style

Its is sometimes more practical to reason about the state of a component rather than its events, and some authors have suggested using a functional programming style to describe this: one reads the "value" of a toggle button by "calling" it, for instance. This can be reproduced using data flow. For instance, consider a toggle button that we want to use as a light switch: instead of reasoning about `on` and `off` events, one would rather reason about the state of the system, as below. Note how we export the state of the toggle and the light, like we did for the dialogue box.

```
component mytoggle {
  rectangle r (0, 0, 100, 50);
  fsm f {;
    state off, on;
    transition (off, r/press, on);
    transition (on, r/release, off);
  }
  switch s {
    color off (black);
    color on (white);
  }
  connector (f/state, s/state);
  value state;
  connector (f/state, state);
}
component mylight {
  value state;
  switch s {
    color on (yellow);
    color off (gray);
  }
  connector (state, s/state);
}
connector (mytoggle/state, mylight/state);
```

Managing modalities and access to resources

After focusing on classical scenarios of graphical user interfaces, we describe here a few scenarios related to new interaction modalities or other computer resources.

Using alternative input

Tablets have an accelerometer to measure their orientation. The following pseudocode shows how one can use it in data flows. Here, we use it to control which part of a graphic scene loaded from a SVG file is visible through a clipping area. This illustrates how much the control model is independent from the event sources, whether they are classical input devices, new input devices, or any other compatible component.

```
component myapp {
  load (file://scene.svg);
  clip c (0, 0, 100, 100);
  connector (accelerometer/x, c/x);
  connector (accelerometer/y, c/y);
}
```

Animation as components

Since all software entities, including data and control structures, are components, animation can also be expressed as components. For instance, in the pseudocode below we modify the application above so that the accelerometer does not control the horizontal position of the clipping zone but the way it drifts over time in an approximation of gravity.

```
component myapp {
  ...
  component xdrift {
    value x;
    value dx;
    clock c (100);
    incr i (x, dx);
    binding (c, i);
  }
  connector (xdrift/x, clip/x);
  connector (accelerometer/x, xdrift/dx);
  ...
```

Note how the combination of connectors and bindings merges two asynchronous input sources (clock and sensor), and ensures multimodal fusion.

Interacting with the context

Future interactive software will not be limited to graphical interfaces. It will react to all sorts of sensors in the user's environment, whether physical or logical. Already, sensors are multiplying and applications are reacting to more and more context changes. To illustrate how this can be addressed with the control structures that we have described, the following code reacts to file creations in a directory.

```
directory d (file://data);
beep b;
binding (d/new, b);
```

Data flow can also be used to similar purposes. For instance, assuming that directories have a `copy` child that copies files when it receives references to them, the following code copies files from a directory to another as soon as they are created.

```
directory d1 (file://data);
directory d2 (file://backup);
connector (d1/new, d2/copy);
```

Not only does this show how control structures originally applied to graphical user interfaces can apply to other types of interactive software. It also illustrates how more and more applications are becoming interactive in the general sense, that is they react to their environment.

CONCLUSION AND PERSPECTIVES

In this article, we have elicited requirements for a framework for control structures in interactive software, described a candidate solution, and provided elements to assess how this solution meets the requirements. The proposed framework constitutes a complete solution to the problem of unifying behavior descriptions in interactive software, and also encompasses control structures used in computation-oriented software. This unification allows programmers to combine control structures at will, and it provides them with an unambiguous model of how their programs are executed.

The djnn programming framework implements the proposed control framework. djnn is available in various languages for instantiating components from the djnn core, its graphical

module and basic input support. djnn is already used in various applications, ranging from an experimental ground station for drones [21] to the control of an interactive showroom and a multimodal aircraft cockpit prototype. Ongoing work includes the formal definition of the execution model, the modeling and implementation of various interaction modalities and bridges to computer resources.

Because it permits to describe programs and their control by solely instantiating components, the proposed framework can serve as the basis for graphical editors and visual languages in addition to textual programming languages. Future research includes the design of a collection of visual and textual programming languages that would each specialized in a different facet of interactive software (graphics, architecture, behaviors, etc) and that could be used by design teams to produce applications together.

REFERENCES

1. Appert, C., and Beaudouin-Lafon, M. Swingstates: Adding state machines to Java and the Swing toolkit. *Software: Practice and Experience 38*, 11 (2008), 1149–1182.
2. Appert, C., Huot, S., Dragicevic, P., and Beaudouin-Lafon, M. Flowstates: prototypage d'applications interactives avec des flots de données et des machines à états. In *Proc. IHM '09*, ACM (2009), 119–128.
3. Backus, J. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM 21*, 8 (1978).
4. Brooks, F. P. The Mythical Man-Month. In *Proceedings of the international conference on Reliable software*, ACM Prss (1975).
5. Chatty, S. Extending a graphical toolkit for two-handed interaction. In *Proceedings of the ACM UIST*, Addison-Wesley (Nov. 1994), 195–204.
6. Chatty, S., Sire, S., Vinot, J., Lecoanet, P., Mertz, C., and Lemort, A. Revisiting visual interface programming: Creating GUI tools for designers and programmers. In *Proceedings of the ACM UIST*, Addison-Wesley (Oct. 2004), 267–276.
7. Colaço, J.-L., Pagano, B., and Pouzet, M. A conservative extension of synchronous data-flow with state machines. In *Proc. of ACM EMSOFT'05*, ACM (2005), 173–182.
8. Coutaz, J., Demeure, A., Caffiau, S., and Crowley, J. L. Early lessons from the development of spok, an end-user development environment for smart homes. In *Proc. ACM Ubicomp'14*, ACM (2014), 895–902.
9. Dix, A. *Research Methods for Human-Computer Interaction*. Cambridge University Press, 2008, ch. Theoretical analysis and theory creation, 175–195.
10. Dragicevic, P., and Fekete, J.-D. Support for input adaptability in the icon toolkit. In *Proceedings of the Sixth International Conference on Multimodal Interfaces (ICMI'04)*, ACM Press (2004), 212–219.
11. Elliott, C., and Hudak, P. Functional reactive animation. In *International Conference on Functional Programming* (1997), 263–273.
12. Fisher, D. A. A survey of control structures in programming languages. *SIGPLAN Not. 7*, 11 (Nov. 1972), 1–13.
13. Gram, C., and Cockton, G., Eds. *Design Principles for Interactive Software*. Chapman & Hall, Ltd., 1997.
14. Harel, D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming 8*, 3 (June 1987), 231–274.
15. Hartson, H. R., and Hix, D. Human-computer interface development: Concepts and systems for its management. *ACM Computing Surveys 21* (1989), 5–92.
16. Jacob, R., Deligiannidis, L., and Morrison, S. A software model and specification language for non-WIMP user interfaces. *ACM Transactions on Computer-Human Interaction 6*, 1 (1999), 1–46.
17. Jacob, R. J. K. Using formal specifications in the design of a human-computer interface. *Communications of the ACM 26* (1983), 259–264.
18. Johnson, R. E., and Foote, B. Designing Reusable Classes. *Object-Oriented Programming 1*, 2 (1988).
19. Lecolinet, E. A molecular architecture for creating advanced GUIs. In *Proceedings of the ACM UIST* (2003), 135–144.
20. Lee, E. A., and Sangiovanni-Vincentelli, A. Comparing models of computation. In *Proceedings of ICCAD* (1996), 234–241.
21. Mathieu Magnaudet, M., and Chatty, S. What should adaptivity mean to interactive software programmers? In *Proc. ACM EICS'14*, ACM (2014), 13–22.
22. Milner, R. Functions as processes. *Mathematical Structures in Computer Science 2*, 2 (1992), 119–141.
23. Myers, B. Separating application code from toolkits: Eliminating the spaghetti of callbacks. In *Proceedings of the ACM UIST*, Addison-Wesley (1991), 211–220.
24. Myers, B. A. A new model for handling input. *ACM Transactions on Office Information Systems* (July 1990), 289–320.
25. Myers, B. A., et al. Garnet, comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer* (Nov. 1990), 71–85.
26. Myers, B. A., and Rosson, M. B. Survey on user interface programming. In *Proceedings of ACM CHI'92*, ACM (1992), 195–202.
27. Navarre, D., Palanque, P., Jean-Franois Ladry, J.-F., and Barboni, E. ICOs: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. *ACM TOCHI 16*, 4 (Nov. 2009), 18:1–18:56.
28. Ungar, D., and Smith, R. B. Self: The power of simplicity. In *Proceedings of the ACM OOPSLA* (Oct. 1987), 227–241.