

Djnn/Smala: A Conceptual Framework and a Language for Interaction-Oriented Programming

Mathieu Magnaudet, Stéphane Chatty, Stéphane Conversy, Sébastien Leriche, Célia Picard, Daniel Prun

▶ **To cite this version:**

Mathieu Magnaudet, Stéphane Chatty, Stéphane Conversy, Sébastien Leriche, Célia Picard, et al.. Djnn/Smala: A Conceptual Framework and a Language for Interaction-Oriented Programming. Proceedings of the ACM on Human-Computer Interaction , 2018, Proceedings of the ACM on Human-Computer Interaction, 2 (EICS), pp.1 - 27. 10.1145/3229094 . hal-01815222

HAL Id: hal-01815222
https://enac.hal.science/hal-01815222

Submitted on 10 Dec 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Djnn/Smala: A Conceptual Framework and a Language for Interaction-Oriented Programming

MATHIEU MAGNAUDET, STÉPHANE CHATTY, STÉPHANE CONVERSY, SÉBASTIEN LERICHE, CELIA PICARD, and DANIEL PRUN,
École Nationale de l'Aviation Civile, France

The persistent difficulty to develop and maintain interactive software has unveiled the inadequacy of traditional imperative programming languages. In the recent years, several solutions have been proposed to enrich the existing languages with constructs dedicated to interaction. In this paper, we propose a different approach that takes interaction as the primary concern to build a new programming language. We present Djnn, a conceptual framework based on the concepts of process and process activation, then we introduce Smala a programming language derived from this framework. We propose a solution for the unification of the concepts of event and data-flow, and for the derivation of complex control structures from a small set of basic ones. We detail the syntax and the semantics of Smala. Finally, we illustrate through a real-size application how it enables building all parts of an interactive software. Djnn and Smala may offer designers and programmers usable means to think of interactions and translate them into running code.

CCS Concepts: • **Software and its engineering → General programming languages**;

Additional Key Words and Phrases: Interactive Software ; GUI Programming ; Reactive Programming ; Djnn ; Smala

1 INTRODUCTION

Interactive software is widespread, from our daily-used smartphone's applications to the most critical systems such as the Human-Machine Interface (HMI) of an air traffic control position or that of a medical equipment. All these systems share a common feature: the need to continuously react to a variety of heterogenous sources of events. Yet, despite their pervasiveness, they are still notoriously difficult to conceive and maintain. If part of this complexity may come from the system design itself, an accidental complexity may also be induced by the inadequacy of the engineering tools, especially programming languages. As a matter of fact, most of them have been firstly conceived to ease the programming of algorithms i.e. the specification of an ordered sequence of instructions to compute a result from an input value. But programming interaction introduces new challenges that are reputed more complex than computation [51].

In the last decades, progress has been made to ease reactive programming noticeably by introducing new event-based facilities in object-oriented languages such as C# events, JavaFX properties or Qt signal/slot mechanism. In the meantime, some work has been done to introduce the data-flow concept in the functional paradigm [18, 33]. More recently, some proposals have been made toward the unification of object-oriented events and reactive functional programming [16, 44], thus attempting to combine their respective advantages. Nevertheless, none of these approaches completely reach the goal of simplification. On

Authors' address: Mathieu Magnaudet; Stéphane Chatty; Stéphane Conversy; Sébastien Leriche; Celia Picard; Daniel Prun, Université Fédérale de Toulouse, École Nationale de l'Aviation Civile, 7 avenue Edouard Belin - BP 54005, Toulouse Cedex 04, 31055, France, firstname.lastname@enac.fr.

© 2018

the one hand, at the semantic level, by combining paradigms they force the programmer to juggle with multiple heterogeneous abstractions (object, function, signal, event), and multiple execution models (sequence of instructions, automatic updating of properties, event listeners). On the other hand, at the syntactic level, they struggle to provide constructs allowing to easily express relationships between events and reactions.

Here, we propose a different approach. Building upon the concept of process inherited from the so-called synchronous languages [6, 7, 19], we propose to extend its use both for the design of control structures governing the execution of a program, and for the modelling of the various parts of an interactive software. Process is roughly defined as change, or as "something that is going-on": a finger moving on a screen, a calculation, the arrival of new data through the network, etc. The goal is to elaborate a rich-enough concept of process so as to embrace all needs of interactive applications (rich graphics, multiple input devices, networking etc.) We thus expect to reach simplification by unification, thus the reduction of the number of basic concepts, rather than by their composition.

The contributions of this paper are the following:

- We describe a conceptual framework, named Djnn based on the concepts of process and causal relationships between processes. More specifically, we show how these concepts are effective at both unifying event and data-flow, and deriving complex reactive control structures such as a Finite State Machine.
- We introduce Smala a new programming language implementing this conceptual model. We show how its syntax, equipped with causality operators, allows for a concise expression of the relation between events and reactions.
- We demonstrate through a real-size application, the HMI of a conventional all-electric helicopter, how Smala supports the development of a non-trivial interactive system.

The paper is organised as follows. In Section 2, we illustrate through an example the difficulties raised by the development of interactive systems. In Section 3 we present the new conceptual model that we propose and we show how it addresses the various aspects of an interactive application. Then, in Section 4, we present Smala, the principles of the language, its syntax, its semantics and its implementation. Finally, we detail its use for the development of a real-size application in Section 5, before reviewing some related work in Section 6.

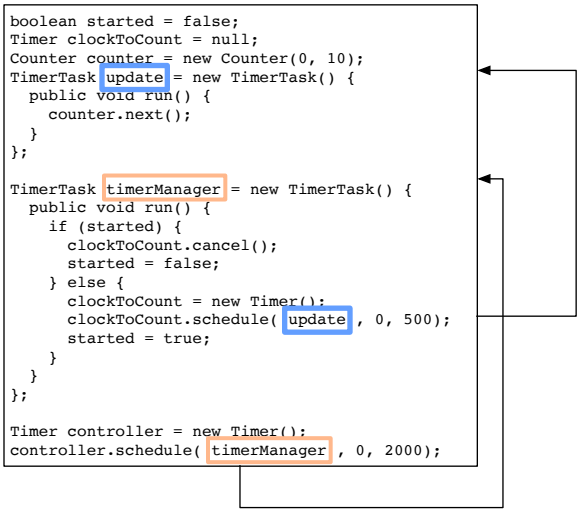
2 PROBLEM STATEMENT

To grasp the problem we want to address, we propose to detail a scenario inspired from the classic counter example that one can find in the literature on reactive systems. Suppose that one wants to build a program that increments a counter on each tick of a clock, and to control the connection between the clock and the counter so that it is sometimes active, and sometimes not. The control over the connection is made by another timer that regularly activates and deactivates the counting process. Such a scenario allows us to explore the facilities provided by a peculiar programming framework as for the programming of interactive software. More specifically, one may assess how it helps the programmer in the task of connecting an asynchronous event source to a specific reaction of the application.

2.1 Callbacks and the observer pattern

A classic solution when working with events in a sequential imperative language is to use either a callback mechanism or the observer pattern. Both consist in defining a function or a method in an object-oriented language, and to subscribe to an event source in such a

way that each time an event occurs, a piece of code is executed. Below (fig. 1) is a possible implementation in JAVA of the proposed example using the callback mechanism.



```
boolean started = false;
Timer clockToCount = null;
Counter counter = new Counter(0, 10);
TimerTask update = new TimerTask() {
  public void run() {
    counter.next();
  }
};

TimerTask timerManager = new TimerTask() {
  public void run() {
    if (started) {
      clockToCount.cancel();
      started = false;
    } else {
      clockToCount = new Timer();
      clockToCount.schedule( update , 0, 500);
      started = true;
    }
  }
};

Timer controller = new Timer();
controller.schedule( timerManager , 0, 2000);
```

Fig. 1. Controlling a counter with a callback (JAVA) entails complex code entanglement.

Many criticisms have already been made to this kind of solutions [32, 35], the main one being that it leads to a complex entanglement of code as soon as the execution of the callback entails the subscription to another one. As it appears in this example, the order of the written program does not reflect the order of the causal relationship between the expiration of the timer and the executed action. To understand the program one has to go back and forth, from the subscription to the instruction to be executed.

2.2 Beyond the spaghetti of callbacks

In order to tackle this issue, several solutions have been proposed. Some of them rest on the introduction of functional reactive constructs in an object-oriented language [16, 44]. This solution has the advantage to preserve the causality order in the writing of a program. By using the latest features of the JAVA language that allow the use of lambda expressions, one can translate the previous example in the code of fig. 2.

```
MyTimer t;
Counter counter = new Counter(0, 10);
MyTimer timerManager = new MyTimer();
timerManager.schedule(() -> {
  if (started) {
    t.cancel();
    started = false;
  } else {
    t = new MyTimer();
    t.schedule(() -> {
      counter.next();
    }, 0, 500);
    started = true;
  }
}, 0, 2000);
```

Fig. 2. Controlling a counter with lambda expression (JAVA) preserves the causality order.

This kind of solutions has been criticised because they were not fixing the inversion of control inherent to this mechanism [44]. This led to the introduction of new programming constructs such as reactors and the *loop*, *loopUntil* and *await* methods that act like event-based control-flow statements [44].

```
Reactor.loop { self =>
  // step 1
  val path = new Path((self await mouseDown).position
  self.loopUntil(mouseUp) { // step 2
    val m = self awaitNext mouseMove
    path.lineTo(m.position)
    draw(path)
  }
  path.close() // step 3
  draw(path)
}
```

Fig. 3. Getting rid of the inversion of control in REACT.SCALA (figure from [44])

However, such statements do not lend themselves to elegant solutions with interactions slightly more complex than the simple "drag and drop" demonstrated in [44] (Figure 3). For example, a "drag and drop" with hysteresis [15] cannot be described with a single "await-loopUntil-await" sequence since a release event may shorten the interaction before a sufficient movement triggers a drag operation. This is even truer with interactions that go back and forth between multiple states (e.g. resizing a rectangle and switching on and off the preservation of the proportions with the shift key): such transitions between states cannot be reduced to a linear control-flow. This suggests that other constructs are necessary to handle those situations.

In the following, we intend to explore a different path noticeably by embracing state-change as a construct that is desirable in certain situations. This is a major difference with frameworks based on a functional paradigm that purposely hide states or get rid of state changes. Most importantly, we wanted to explore a language with no attempt to embed it seamlessly in a host one (e.g. Java). Of course, this has the harmful consequence that constructs from the host language cannot be reused, but this offers us more freedom in the design of our language, and possibly more original and interesting results than if we had inherited constraints from the host language.

3 THE DJNN CONCEPTUAL FRAMEWORK

3.1 A process-based approach

Programming paradigms are often grounded on a metaphor or encourage a particular way to model the world. For example, the object-oriented paradigm found some of its inspiration in the philosophical literature [29]. The concept of object instantiation thus offers an illustration of the relation between the abstract ideas and the concrete objects as expressed in the Platonician ontology. In a more general way, the concepts of inheritance and composition, as well as those of attributes of an object, embed a certain vision of the world as a complex organisation of things having properties (mass, position, velocity, etc.). But this so-called substantialist ontology is not the only way to envision the world. Another major ontology, that can be traced back to Heraclitus, takes as its primary building block the concept of process. The basic idea here is to consider the world as a matter of change and to focus on what is going-on [42, 46].

This process ontology is amenable to a formal theory defining, for example, types of processes and compositing rules. Moreover, the concept of causality finds a nice interpretation

in a process ontology, as a specific interference between processes [43, 45]. If we agree that programming interaction is very much a matter of programming causality [27], then a process ontology appears as a well-fitted conceptual framework.

3.2 Process modelling

Processes have been mainly theorised in computer science by Hoare [22] and Milner [34] with the aim of providing concepts and formalisms for modelling systems that interact with their environment. Their focus was on the formalisation of the synchronisation and the communication between processes that run concurrently. According to their view, a process is a description of a behaviour, something that is going on, such as the famous vending machine that delivers a chocolate when a coin is inserted. Following this idea, we propose to take the concept of process as the simplest expression of change, and to model interactive software as a set of causally connected processes.

As an illustration, consider the process of moving the pointer over the display. It can be broken up in several sub-processes, each one triggering the next. When the mouse moves on the desktop, it triggers the sending of a USB frame. The arrival of a USB frame triggers a decoding process by a series of drivers. Then, *dx* and *dy* values are transmitted by the Operating System to the windowing system that triggers a process of accumulation and possibly clipping to get a new position for the pointer. Finally a process of redrawing is executed. At the abstract level, this is a description of a causal chain between processes i.e. a set of relationships stating that when something happens, then something else must happen (Figure 4).

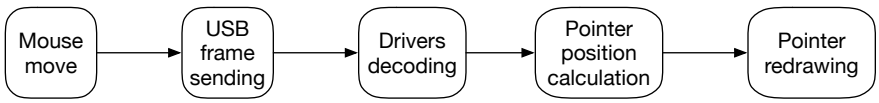


Fig. 4. Decomposition of the pointer move process

Processes can be generalised and distinguished by the specific action they complete. Some processes are pure computation, others are drawing operations, sound synthesis, network communication, etc. At a finer grained level, one can also consider as a process the simple assignment of a new value to a variable, or a property as the process of maintaining and monitoring a value in a memory cell.

We make two further statements about processes. Firstly, a process may have a structure: it can be made up of several sub-processes. An adder, for example, can be described as the composition of three properties, `left`, `right`, `result`, and an adding process triggered each time `left` or `right` are modified. In the same way, a rectangle can be modelled as the composition of four properties, `x`, `y`, `width`, `height`, and a redraw process triggered each time one of the above properties is changed. Secondly, we consider processes as having an activation status. A process can be active or inactive. When activated, it produces its specific behaviour.

From this, we can define a causal relationship between processes as the propagation of activation from one process to another one. This characterises a basic control structure from which other ones can be built.

3.3 Controlling processes

At the most general level, a control structure is a construct that specifies an activation ordering among processes. Let us call 'coupling' the simplest control structure, defined as the unidirectional activation relationship between processes. If two processes A and B are coupled, then when A is activated it triggers the activation of B.

The data-flow concept, sometimes named signal or behaviour, i. e. a reaction to a stream of values, can be built upon this relationship. As suggested above, we define a property as a process of maintaining and monitoring a value. Each time the value of the memory cell is changed, the property propagates an activation flow toward previously coupled processes, if any. An assignment is a process whose defining behaviour is to copy a value from one property to another one. A data-flow can thus be defined as the coupling of an input property with an assignment process to an output property: each time the input property is modified, the assignment process is activated which results in the value of the property being copied to the output property.

Regarding the concept of event, we propose to define it in a very general way as the activation of a process. Some processes are activated by external sources such as a mouse button press or an embedded timer. Others are internal to the program such as the activation of an addition or the activation of a redraw process.

The important point is that, by defining the concept of data-flow and the concept of event over the concept of process, we achieve the wanted unification of both concepts [44]. A data-flow is the composition of two coupled processes (i. e. two coupled events): a property monitoring a value and an assignment.

More complex control structures can also be derived from this basic 'coupling' relationship [10]. A finite state machine for example is a complex process made of states and transitions between states. A transition may be defined as a coupling between two states. A state is itself a process whose activation consists in the activation of a list of transitions i.e., couplings between processes considered as source and destination states of the transition. When a source is activated, the activation is propagated to the new state. The previously active state and transitions are deactivated and the new ones are activated.

The figure 5 summarizes the basic concepts of the Djnn's framework.

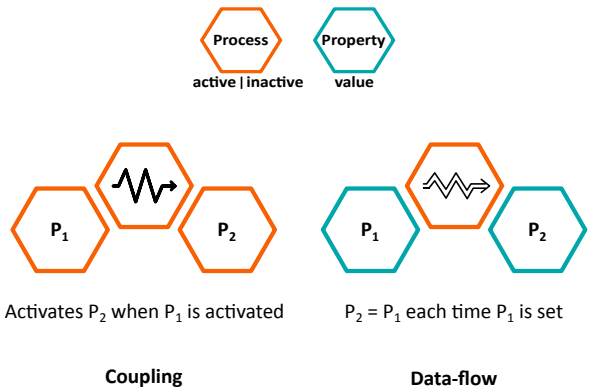


Fig. 5. Basic concepts of the Djnn's framework.

4 THE SMALA LANGUAGE

Based on this conceptual framework, we have designed SMALA[1], a new programming language that takes as its primary concern the development of interactive applications. The design of this language intends to satisfy three major requirements:

(1) Conceptual unification. We make the hypothesis that the lower the number of basic concepts, the easiest will be the learning of the language. Such a unification is provided by the process-based conceptual framework described above.
(2) Interaction-oriented programming. The conceptual model and the syntax must efficiently support the specification of interaction and in peculiar of causal relationships between processes. Thus the expression stating the relation between an event source and an action should be kept as short as possible. It should be visually salient[15], and should preserve the order of the causality.
(3) Support for iterative design. It is a known fact that designing user-centred systems should be done incrementally and iteratively, because the requirements change when the users are confronted to the system in development [24]. The ability of a programming language to support such a process can be measured by its ability to support change in graphics or in event source without a complete refactoring of the code.

In the following, we propose to introduce SMALA through few examples, then we present its syntax and its basic semantics.

4.1 Programming with Smala

4.1.1 Example 1. The following code increments a counter on each tick of a clock. Its behaviour is equivalent to that of the JAVA examples described in section 2.

```
Component root {
  Clock cl (500)
  Counter count (0, 1)
  cl.tick -> count.step
}
run root
run syshook
```

This short program illustrates the declarative style of programming chosen for SMALA. Firstly, we declare a parent node, `root`, to which we attach some children, a clock `cl` and a counter `count`. The last child is a causal link between the previous ones declared through the special symbol `->`. We call this component that builds an explicit coupling between nodes a 'binding'.

Once the tree has been declared, it can be started. This is done by the instruction `run root`. The start of a component is propagated to its children, in a depth-first order. The last instruction starts a special component, named `syshook`, that creates the link with the underlying Operating System. This is required as the OS manages in this case the timers on which the clock is built.

The `tick` node, accessed though a dot notation, is a built-in child of the clock component provided by SMALA. Once the clock is started, `tick` is periodically activated. The duration of the period is specified by the given parameter in milliseconds. On its side, the counter has a built-in child named `step` as well as a child named `output` that encapsulates a numerical value. Each time the `step` is activated, the `output` is incremented by the `delta` value given as the second parameter of the counter. Thus, the binding component acts as an activation

[1] http://smala.io

relay between the clock's `tick` and the counter's `step` that triggers the periodic increment action.

Despite its brevity, this example illustrates two important benefits of the language and its underlying model: firstly, it preserves the causality order and makes explicit the direct relation between processes, here the clock's tick and the counter increment. Secondly, by generalizing the concepts of process activation and coupling between processes, it allows the writing of programs in a very concise way.

4.1.2 Example 2. The next example implements a data-flow between two properties. This is a classic adaptive (or responsive in web terms) example where we want to modify the size of a graphical object according to the size of its container (Figure 6).

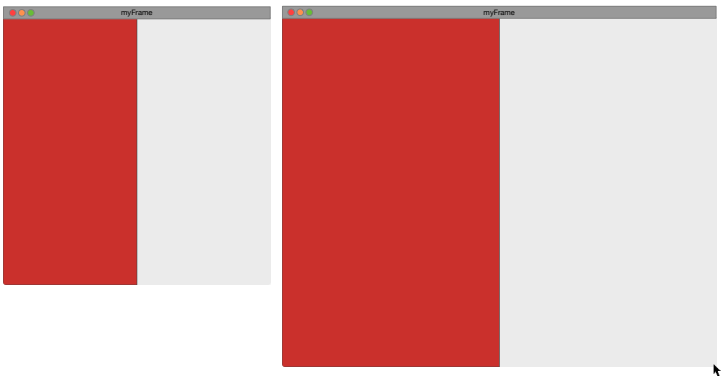


Fig. 6. Propagating the values of the frame's geometry to the rectangle

```
Component root {
  Frame f ("myFrame", 0, 0, 500, 500)
  FillColor fc (200, 50, 50)
  Rectangle r (0, 0, 150, 500, 0, 0)
  f.height => r.height
  f.width / 2 => r.width
}
```

In this example, the tree of nodes is made of three graphical components: a frame, a fill colour, and a rectangle. Note that the colour is not a property of the graphic shape: it is a full-fledge component that works as a modifier of the implicit graphic context. Two control structures link the size properties of the frame to those of the rectangle. This is done through the dedicated symbol => which denotes a component named 'connector'.

A connector is a component that implements the concept of data-flow. Each time the left side of the operator is modified, its value is copied on the right side. The left side can be any logical or arithmetic expression built from a combination of Smala properties and literals. As usual in data-flow programming, the expression is re-evaluated each time one of its properties is modified. The two connectors of this program ensure that the rectangle will have the same height and half the width of the frame.

This example shows the strong similarities, at the syntax level, between event-based reaction and data-flow within Smala. In both case, this is expressed as a simple operator relating two components.

4.1.3 Example 3. The last example highlights more advanced features of SMALA: complex control structures and graphic integration. This program displays a button and triggers a sound each time the button is clicked.

```
Component root {
  svg = loadFromXML ("file:///img/button.svg")
  Frame f ("myFrame", 0, 0, 500, 500)
  Beep action (1)
  Switch button (idle) {
    idle << svg.idle
    pressed << svg.pressed
  }
  mask << svg.mask
  FSM fsm {
    State idle
    State pressed
    idle->pressed (mask.press)
    pressed->idle (mask.release, action)
    pressed->idle (f.release)
  }
  fsm.state => button.state
}
```

When building HMIs, as soon as the required graphics becomes complex, it is far easier to use a dedicated tool such as Inkscape or Illustrator™. SMALA provides facilities to load a tree of components stored in XML format and, more specifically in the SVG format (an XML instance) provided by those tools. Line 2 of the above program loads an SVG file containing the various graphical appearances of a button. Parts of this loaded tree can be added to the current tree. This is done through the dedicated symbol `<<`. At line 6, for example, the child node `idle` of the previously loaded `svg` component is added to the current tree and is given the name `idle` (the name does not need to be the same). Hence, this allows a complete redesign of the architecture of the graphics part without any code modification, provided that the newly produced SVG file has a child node named `idle`.

In this program, parts of the SVG are added to a control structure named a 'switch'. The defining feature of this type of component is that only one of its children node can be active at a time. This means that either `idle` or `pressed` can be active during execution. Changing the active child in a `Switch` is done by changing its `state` text property to the name of the desired child. When the value of `state` changes, the `Switch` component resolves the provided name, stops the current active child, and activates the desired child. If no corresponding name is found, no child is activated. In this example, the switch's state is driven by the state of a finite state machine (l.17).

The declaration of an FSM is classically done through the declaration of a list of states and transitions between states. The optional `action` parameter (l.14) allows to specify a Mealy machine, that is, the action is activated during the transition, before the activation of the destination state [2]. Here, the transitions are triggered by mouse events coming from the Operating System, routed to built-in children of graphical shapes by the execution engine.

[2] By difference, a Moore machine emits its output when it reaches the destination state. Both automaton are known to be equivalent and both has been used for UI development [14], however a Moore machine has more states and transitions which may add some complexity.

4.2 Syntax

A SMALA program is a description of a tree of components. Components are declared through the specification of a type followed by an id and possibly some parameters.

```
Int i(0)
Rectangle r (0, 0, 50, 50, 0, 0)
```

All components have a type which specifies their behavior when they are activated. However, all components belong to the most generic type Component. This is a crucial point as the control structures are defined between generic components. Control structures are themselves components that can be activated or deactivated. Some types of components can contain sub-components. The parent-child relationship is specified through the use of brackets. In the following code, the children of a predefined Switch are declared inside a pair of brackets:

```
Switch sw {
   FillColor red (255, 0, 0)
   FillColor green (0, 255, 0)
}
```

A subcomponent can be accessed from the outside through its path (e.g. `sw.red`). A component and its structure can be declared and instantiated at the same time with the `Component` keyword:

```
Component Button {
   Rectangle r (0, 0, 50, 50, 0, 0)
   Text t (10, 10, "ok")
}
```

In order to reuse the structure of a component, one can define it in a separate file, prefix it with the `_define_` keyword, and suffix it with a list of parameters:

```
// Button.sma
_define_
Button(String text, int w, int h,
            Component action) {
   Rectangle r (0, 0, w, h, 0, 0)
   Text t (10, 10, text)
   r.release->action
}
```

To use it, one has to import it, and instantiate it like any other existing Component. Basic types such as `Int`, `Bool`, `String`, `Float` are passed to the constructor by value, while `Component`s are passed by reference:

```
// main.sma
import Button
...
  Component do_it
  Button b ("myButton", 100, 20, do_it)
...
```

To facilitate abstraction and hide implementation, SMALA provides the `aka` keyword (also known as). This makes a component available to the outside through a new path without instantiating a Component:

```
// Button.sma
_define_
Button(String text, int x, int y,
           int w, int h, Component action) {
   Group g {
```

```
        Translate t (x, y)
        Rectangle r (0, 0, w, h, 0, 0)
        Text label (10, 10, text)
    }
    g.r.release->action
    x aka g.tx
    width aka g.r.width
}

// main.sma
import Button
Component root {
  Beep do_it
  Frame f ("myFrame", 0, 0, 500, 500)
  Button b ("button", 100, 20, do_it)
  f.width - b.width - 10 => b.x
}
```

As seen above, changing values are done thanks to the assignment operator `=:`. The assignment is activated once only when its parent component is activated, while the connector `=>` is continuously pushing values once activated. The simpler `=` is reserved for initialisation before the execution. We borrowed the `=:` lexeme from Algol, but reversed it to be consistent with the connector/data-flow operator `=>`. This makes it easier to read and to modify an assignment to a data-flow and vice-versa.

Some components are provided by the environment such as input devices, displays, network interfaces. They are provided as global identifiers that denotes sets of components. The identifier `TouchPanels` for example denotes the set of touch devices plugged to the computer. Sets have two buit-in children `$added` and `$removed` that are activated each time an item is respectively added and removed from the set. These children are properties whose value is a reference to the concerned item.

Besides the purely declarative part, SMALA provides some facilities to handle the tree of components. It is thus possible to add children to a specific component with the instruction `addChildrenTo` and to add a name pointing to a specific component in the current context with the instruction `find`. Moreover, it is possible to initialize the value of a property before the execution of the tree with the equal `=` symbol. The following instruction:

```
counter.state = 10
```

assigns the value 10 to the property `state` before the activation of the counter.

The main part of the syntax is summarized in figure 7.

On may note that there is no explicit operator to express parallelism. This is explained by the fact that SMALA is not a sequential language. A SMALA program is a specification of what are the active processes, and what are the conditions for the activation and the deactivation of a set of processes. This is the role of the execution engine to guarantee that if several events arrive at the same time then each dependant processes will be activated.

4.3 Semantics

This section introduces the description of the main characteristics of Smala semantics. After introducing the definition of a SMALA program model, and some useful preliminary definitions, we formally define the central structural element of Smala: *Node*. Then, we show how it can be used to define fundamental basic elements.

Expression	e	Classical unary and binary expressions
Identifiers	$id \in$	`String`
Types	$t \in$	{ `Int` , `Double` , `String` , `Component` }
	t_c	Container
List of Components	$C ::=$	$C\ c$
Component	$c ::=$	$t_c\ id\ P$ `{` C `}` $\mid t\ id\ P \mid$ $id \rightarrow id \mid e =: id \mid e \Rightarrow id \mid$ `addChildrenTo` id `{`C`}` $\mid$ id `<<` $id \mid id$ `=` `find` `(` id `)`
List of Parameters	$P ::=$	$P\ p$
Parameter	$p ::=$	$id \mid t \mid e$

Fig. 7. Basic syntax of SMALA

A SMALA program is a tupple (N, A, V) where N is a finite non-empty set of components, A a set of links between elements of N and V the values of some elements of N called properties.

The primitive elements of the language are properties ($Prop$), bindings ($\rightarrow$), assignments ($=:$) and comparators ($==$). All these are components, i. e., part of N. Roughly, a component can perform some action (computation, monitoring of a memory slot, handling inputs and/or outputs with the environment...) and can manage activation. The activation function ($\mathcal{F}^{act}$) models how the control is propagated towards components of the system. Thus, a component has a state of activation whose value is in $\{stopped, started, propagating, propagated\}$. We call $n.state$ the state of component n.

$A \subset N \times N$ is a finite set of arcs representing the structural relationships between elements of N. (N, A) is an ordered tree. $<_A$ is a binary relation that defines a total order over N according to a depth-first left-to-right (N, A) traversal.

We define $Children(n)$ as the set of all the components children of the component $n \in N$ in the (N, A) tree. $Succ(n)$ is defined recursively as the set of all the descendants of a component $n \in N$ in the (N, A) tree.

$$Children(n) = \{n' \in N \mid (n, n') \in A\}$$
$$Succ(n) = Children(n) \bigcup_{n' \in Children(n)} Succ(n')$$

We identify by $min(S)$ the smallest element of set $S \subset N$ relatively to $<_A$.
We call $root$ the smallest element of a SMALA program: $root = min(N)$.

At an operationnal level, a SMALA program is made of components. Those components are instances of sub-types of a generic type definition that represents the nodes of the (N, A) tree: $Node$.

Node. An element of type $Node$ is defined by its name (id), a trigger for the propagation of activation ($trig$), an action (f) and a propagation ($dact$). Roughly said an action is a process that explicitly defines what the defined element performs, possibly nothing. A propagation is a rule of control propagation towards other components of the system. There may be no propagation[3].

[3]To express the definition of the semantic, we use a standard inference rule notation where hypothesis and conclusion are seprated by an horizontal line. Hypothesis are expressed in the upper part and the conclusion

$$\text{definition of } Node\ \frac{id \qquad f \qquad trig \qquad dact}{id[f, trig, dact] : Node}$$

The propagation of activation towards other components of the system occurs when the component is triggered and started:

$$\text{triggering}\ \frac{\forall f\ \ dact, C[f, true, dact] : Node \qquad C.state = started}{dact}$$

Upon activation ($\mathcal{F}^{act}(C)$), the internal action is performed:

$$\text{activation}\ \frac{\forall f\ \ trig\ \ dact, \mathcal{F}^{act}(C[f, trig, dact]) \qquad C.state = started}{f}$$

Propagation of activation between instances. At the creation of the system, the value of the states of all components is *stopped*. Then, the initial `run` (see Section 4.1) turns these states to *started*. The following paragraph describes the propagation that occurs upon activation of a component. This activation may be internal (the initial activation for example) or external (mouse click for instance).

When a component n is activated, then all components whose father or source is n are successively activated according to a depth-first-left-to-right traversal path. During the activation, the state of n is set to *propagating* in order to avoid introduction of loop (activation is propagated to components in *started* state only). Once the activation propagated, the state is set to *propagated*.

We define $Z(n)$ as the set of components whose state is *started* and whose father or source is n. In other words, n is in a direct causal relationship with the elements of $Z(n)$.

$$Z(n) = \{n' \in N \mid S(n') = started \wedge (n' \in Children(n) \vee n = Source(n'))\}$$

The propagation of activation is defined by the following inference rules:

$$\mathcal{F}^{act}(n)\ \frac{n \in N \qquad n.state = started \qquad Z(n) \neq \varnothing}{n.state = propagating; \mathcal{F}^{act}(min(Z(n)))}$$

$$\mathcal{F}^{act}(n)\ \frac{n \in N \qquad n.state = started \vee propagating \qquad Z(n) = \varnothing}{n.state = propagated}$$

Once the activation has been fully propagated, the state of the affected components is set back to *started*. The rules are on the same model as the previous ones. They are not detailed here.

Property. The type property *Prop* defines elements p associated to a value $\nu(p) \in \mathbb{N}$. At any time, this value can be updated to a new one denoted $\nu'(p)$. The update of value $\nu(p)$ triggers the activation of property $p \in Prop$.

$$p: \quad Prop[null,\ \nu(p) := \nu'(p),\ \mathcal{F}^{act}(p)]$$

Binding. A binding between two components c (called the source) and d (the destination) is a component that activates the destination component when the source component is activated.

$$c \to d: \quad \to [null,\ \to .state = propagating,\ \mathcal{F}^{act}(d)]$$

We have used earlier the *Source* function. It is defined as follows: $\forall c \in N\ \forall d \in N, Source(c \to d) = c$. Similarly, $Destination(c \to d) = d$.

at the bottom. In this notation, when the hypothesis are all verified, the conclusion is true. Hence, combining those rules allow to describe a program. e of type T is written as $e : T$. $e[e_1, ..., e_n]$ is a constructor of an element named e with $e_1, ..., e_n$ elements.

Assignment. An assignment between two elements of *Prop* a and b is a component that, when activated, is able to perform the copy of the value of a to b.

$$a =: b : \quad =: [\nu(b) := \nu(a),\ true,\ null]$$

Comparator. A comparator between elements of *Prop* a and b is a component that is triggered by the activation of a or b when their values are identical. This results in the activation of the component.

$$a == b : \quad == [null,\ (a.state = propagating \vee b.state = propagating) \wedge \nu(a) = \nu(b),\ \mathcal{F}^{act}(a == b)]$$

Connector. A connector between two elements of *Prop* a and b is a component that copies the value of a to b each time the value of a is updated and activates b. It can be written combining an assignment and a binding:

$$a \Rightarrow b : \quad a \rightarrow (a =: b)$$

More complex components can be defined combining these previous basic components. In the same fashion we have defined the connector as a combination of an assignment and a binding, the switch or the finite state machine components can be defined as the combination of properties, bindings and comparators.

This semantic contributes directly to the achievement of two of the major requirements identified in Section 4:

(1) Conceptual unification: the semantic is minimal since it is rougly composed of only one element, the *Node*, derived and composed into more complex ones (for instance, binding, assignment, comparator, connector).
(2) Interaction-oriented programming: the semantic explicitly defines causal relationships between components (with bindings and father-children relationships).

4.4 Implementation

Smala is built on the top of a set of C libraries named Djnn[4]. Djnn provides a core library that implements the execution engine allowing to run a tree of components. Once the tree is loaded and started, the core library starts an event loop that manages the events coming from the environment. On arrival, events are dispatched to the components of the tree. The control structures contained in this tree specify an activation graph through which the event are propagated. At the moment, the propagation goes depth-first left-to-right in the graph without prior re-ordering. As such it is partially exposed to the so-called glitch issue [4].

Djnn also provides libraries with some basic components. The *Base* library, for example, contains components for arithmetic, logic as well as some complex control structures such as finite state machines and Petri nets. Graphics is provided by the *GUI* library. It contains shapes, style components, and geometric transformations. It is also responsible of the rendering of the graphics on the display. Three rendering engines are available, one based on the Qt toolkit, another one based on Cairo, and a third one based on OpenGL. The choice of a rendering engine is made according to the target system (OS, windowing system, GPU).

It is possible to build a tree of components by directly using these libraries and the C language. However the task is akin to those of writing an abstract syntax tree. Thus, we designed Smala in order to provide a dedicated syntax with specific symbols that helps the programmer visualise the interaction between components. Smala comes with a compiler that translates the Smala program in the C language.

[4]http://djnn.net

5 IMPLEMENTING A REAL-SIZE HMI WITH SMALA



Fig. 8. The Volta cockpit HMI

Volta is the first conventional all-electric helicopter [2], that has accumulated hours of flight in 2016 and 2017. We have developed the Human-Machine Interface (HMI) of the cockpit with SMALA (see figure 8). The HMI displays important information that must be monitored while flying: the number of rotation per minute of the rotor (RPM), the instantaneous consumed power, the battery charge, the temperatures of various power-chain elements, or the bank of the aircraft.

The HMI has been developed concurrently by a programmer and a graphic designer. These two developers first agreed on a SVG tree structure and on the names used in it. Then the two started to work independently. The graphic designer focused on the graphical part and took the time to test multiple graphic design options. Meanwhile, the programmer produced quick-and-dirty graphics, so as to immediately use them for coding and testing the behaviour of the program. Once the first version of the graphics was available from the designer, the programmer was able to substitute it by just changing an instruction in his program. This illustrates the support of iteration from SMALA in the HMI design process (Requirement 3).

The state of the helicopter is implemented with Int and Double properties: RPM, voltages, temperatures etc. The values of the properties are updated by a process that reads the information coming from a CAN bus. Some values of the state are computed from other values. For example, the instantaneous power is the result of a data-flow multiplying the instantaneous current and the instantaneous voltage (see Appendix A.3). Another data-flow computes a predicate that assesses whether the RPM is below a given threshold (see Appendix A.2). The boolean result of this formula is provided through the model.

In order to update the graphics that reflect the data, the state properties are connected to other numerical properties that act as a model for the graphics. For instance, the boolean low_rpm is connected to a state-machine that controls the graphical status of the “low RPM” alarm. The low RPM alarm is a blinking red circle, which is itself coded with a state-machine that toggles between an “on” and an “off” state on clock events (see Appendix A.2). This

illustrates how Smala enables to seamlessly combine multiple control flow mechanisms (here data-flow and state-machine) to specify interactions (Requirement 1 and 2).

Another example is the gauge that displays the instantaneous current and power. The model of the gauge HMI is a numerical property ("input_gauge") that controls the rotation in degrees of the needle, and another numerical property ("input_text") that controls a text string. A mathematical formula translates the input_gauge (in a range from 0 to 360 A) into the rotation (in a range from 0 to 180°), while a formatter translates the input_text into a string which in turn is connected to a textual SVG node.

Other transformations would translate a value into a red-amber-green-amber-red colour scale (RAG status) to reflect the criticality of a level e.g. of the instantaneous current. This is done through a switch that controls the fill colour of an SVG graphics according to a set of threshold values (see Appendix A.3).

The two previous examples demonstrate the architecture agnosticism of Smala that enables the programmer select variants of the MVC pattern [41], here an M-V-VM pattern (Requirement 1).

The HMI also includes 3 pages of parameters. The display of the current page is driven by a state-machine (see Appendix A.4). The transitions of the state-machine are fired thanks to a left-right switch on top of the helicopter physical cyclic stick. The switch is connected to two GPIOs on the embedded platform. For debugging purpose, the transitions can also be fired by a mouse click on two non-visible graphical rectangles. This example demonstrates how one can seamlessly connect two very different sources on the same control flow mechanism (Requirement 1 and 3).

Finally, the cockpit includes a Primary Flight Display (PFD), which is an instrument that depicts the attitude of the aircraft (mainly its rotations according to the three 3D axes). We have designed three variants of the PFD (Annex A.5): "occidental", "russian", and "new". They vary notably according to the way banking is represented: with a rotating horizon (as opposed to a fixed horizon), with a rotating aircraft icon (as opposed to fixed), or with the banking scale on top (as opposed to bottom). For example "occidental" is a rotating horizon + a fix aircraft + a top scale, "russian" is a fix horizon + a rotating aircraft + a bottom scale and "new" is a rotating horizon + a fix aircraft + a bottom scale. There are other dimensions of variations, all described in the three configurations of the state machine in Appendix A.5. Note how the three descriptions differ by the source value and the type of coupling (either assignment or data-flow) only. Actually, as programmers of both this HMI and Smala, we could not achieve to design a language construct that would reduce further the quantity of signs to express these causal relationships. Besides, as HMI designers, we argue that the differences in the three chunks of code completely characterise the differences between the three design variations [23].

6 RELATED WORK

The literature on the architecture of interactive software often splits an application in two parts: a "functional core" that deals with computation and data, and a "user interface" that deals with interaction [5, 40]. The highlighting of the concept of "inversion of control" (IoC) [28] is a good symbol of differences between them. There are also differences in engineering processes: a growing part of interaction-oriented software is directly produced by designers without help from traditional programmers [12, 36].

When the concept of external control appeared [21], event-based programming was proposed as the solution, and this new control structure was implemented with extensions such as callback functions. Programmers started to face what Myers called the spaghetti of

callbacks [35]: whereas the traditional control structures were successful in hiding useless complexity from algorithm programmers and in letting them focus on their design tasks, they instead exposed useless complexity for user interface programmers and obscured their designs.

These new concerns, specific to user interfaces, were not treated in programming languages. Instead, new control structures such as active values, state machines and constraints were introduced in graphics libraries and gradually became their most prominent features, turning them into programming frameworks. State management was one of the earliest identified concerns, first to describe the global dialogue between the user and the program [26], then to describe the behaviors of individual components [37]. Attempts were also made at managing state in combination with another growing concern: the need to organize the complexity of programs, using hierarchical systems [8, 20]. In the same way, Visto [1] is a proposal for a language that adds an object system on top of the Haskell functional language, thus allowing the management of state and behaviour. Other concerns appeared when the interaction styles became more continuous: animation, drag and drop, data visualization. The focus of interest in these situations is more about the flow of information than about the state changes, and various forms of data flow control were proposed to support the description of data representation and graphics layout [38], animation [9], and input [9, 17].

Taking advantage of similarities between data flow and the flow of values in functional programming, some authors have proposed to combine some features of reactive programming paradigm with the syntax of functional languages. This produced functional reactive programming languages [18], which combine events and data flows with a computation-oriented programming syntax. Some of them offer a way of describing reactive programs in a purely functional style [49]. 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 [3, 30], obtaining some form of syntactical consistency. The C# programming language also provides the same consistency for the basic event mechanism.

The above solutions create two levels of control structures: a standard framework to describe for the overall architecture of a program, and specialized constructions for describing some details. Other works take the same approach but reverse the roles; for instance, 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 [39]. 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 [9, 13, 25].

All these are partial solutions; they provide solutions for common situations but still impose constraints, when modern interactive products require more and more flexibility in terms of event types and control patterns. A tighter integration was described in [12], 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.

A recent survey on reactive programming [4] highlights six properties to build a taxonomy of reactive programming approaches and discuss the open challenges in the field. One of them, *Lifting Operations* can be used to illustrate the need and the gap for unification. Lifting operations are needed when reactive programming is embedded in host languages (either as a library or as a language extension), to let existing language constructions have a reactive meaning. In this survey, authors state that lifting must always happen manually in languages that require the explicit tracking of data-flow-dependent values by programmers. To reduce the gap, when the host language supports operator overloading (such as Haskell)

it is possible for the reactive language programmers to offer a large set of lifted operators specifically designed to dedicated problems (such as animation). Finally, it is possible to get implicit lifting on dynamically typed languages but this can be difficult to achieve with widely used languages. For example Flapjax [33], a reactive language for Web programming built on top of JavaScript needs to be pre-compiled with a dedicated tool that applies code transformations to generate lifted operators.

Unification is sometimes claimed to be achieved by deeply integrating different paradigms. For example, the aim of REScala [44] is the unification of imperative, modular events and reactive values to support a mixed oriented-object and functional style in designing reactive systems. REScala is built on top of Scala, a multi-paradigm language designed to support both oriented-object and functional style. The main benefit of this approach is to let the programmer choose among the different styles while still having access to the whole set of existing libraries [50]. We believe that if seamless integration of paradigms might be a powerful solution for advanced and experienced programmers, full unification would be even more beneficial to simplicity. Moreover, we believe that full unification would help in setting up validation and verification tools which would be very difficult to achieve in a multi-paradigm language.

Finally, the need for unification is pointed out in most recent work. For example, in [48] authors identify a number of code characteristics that do not map onto the reactive programming paradigm but that are present in many real life reactive programs. As a solution, the paper describes an actor-based model that can serve as the basis for future language designs that allow a programmer to use what they call "the awkward squad" without making the reactive parts of the program accidentally non-reactive.

7 CONCLUSION

We have presented Djnn, a new conceptual framework for the design of interactive software, and Smala, a programming language based on this framework. The most significant contribution is that we built our approach on a very small number of basic concepts, mainly those of process activation and causal relationship between processes. We showed how the concept of process activation provides a convenient way to unify the concepts of event with that of data-flow. Our approach has few more advantages. First, the reduction of the number of basic concepts might diminish the complexity sometimes induced by the combination of several paradigms. Second, as it has been shown in [31], it allows to model complex interactive software, such as adaptive user interfaces. Finally, it provides a basis for the formal verification of new properties such as the visibility of a graphic object [11].

As regard to the requirements expressed in the beginning of the section 4, one may thus conclude that the first one, the conceptual unification, has been achieved. For the second one, the support for the expression of causal relationships, we also assume that the various provided examples show how the execution model and the syntax of our language make it easier to program interactions. The final one, the support for iterative design, is harder to assess as it relates to a complete engineering process with multiple actors. However, the direct import and change of graphic elements without code refactoring is a major feature that simplifies the transitions from to the early low-fidelity prototypes to the final application.

Regarding the Smala language, we showed how it brings significant support for the development of interactive system. It allows the programmer the concisely express causal relationships between processes, thanks to an arrow-based syntax connecting causes and effects. This arguably reduces the gap between the expression of a desired interactive behaviour and its programming. Moreover, the generalisation of the concept of process

makes easier the iterative development of an interactive software as it allows the substitution of one event source by another one at minimal cost. Finally, it helps the development of visually rich user interfaces by facilitating the collaboration of programmers with graphics designers using their own tools.

Note that, for now, SMALA offers only the very basic building blocks for the programming of a graphical user interface. There is not such a thing as a set of common widgets (buttons, selectors, sliders, etc.). However, we provide all the required graphic elements as well as the control structures allowing to build them from scratch. In the same way, the set of basic components may appear incomplete. For example, we do not provide support for file management or sound design. This will be addressed in the near future.

Future work includes extending the current sets of control-flow structures and providing a better and more robust (if not verified) implementation. In the short term, we plan to enrich the semantics with the description of more complex components composing the primitive components described in this paper. We also plan to extend the set of verifications on HMI that the language enables [11]. To this end, we aim to develop an implementation of the semantics and perform formal verification with the Coq proof assistant[47].

ACKNOWLEDGMENTS

REFERENCES

[1] Kris Aerts. 1999. Visto: A More Declarative GUI Framework. In *Computer-Aided Design of User Interfaces II, Proceedings of the Third International Conference of Computer-Aided Design of User Interfaces, October 21-23, 1999, Louvain-la-Neuve, Belgium*, Jean Vanderdonckt and Angel R. Puerta (Eds.). Kluwer, 73–78.

[2] Philippe Antoine and Stéphane Conversy. 2017. Volta: the first all-electric conventional helicopter. In *Proceedings of the More Electrical Aircraft Conference (MEA '17)*. ACM Press.

[3] Caroline Appert and Michel Beaudouin-Lafon. 2008. SwingStates: Adding state machines to Java and the Swing toolkit. *Software: Practice and Experience* 38, 11 (2008), 1149–1182. https://doi.org/10.1002/spe.867

[4] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A Survey on Reactive Programming. *ACM Comput. Surv.* 45, 4, Article 52 (Aug. 2013), 34 pages. https://doi.org/10.1145/2501654.2501666

[5] Len Bass, R. Pellegrino, S. Reed, R. Seacord, R. Sheppard, and M. R. Szezur. 1991. The Arch Model: Seeheim Revisited. (April 1991). Presented at the CHI'91 User Interface Developers Workshop.

[6] Albert Benveniste and Paul Le Guernic. 1991. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming* 16, 2 (1991), 103–149.

[7] Gérard Berry and Georges Gonthier. 1992. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming* 19, 2 (1992), 87–152.

[8] Renaud Blanch and Michel Beaudouin-Lafon. 2006. Programming rich interactions using the hierarchical state machine toolkit. In *Proceedings of the working conference on Advanced visual interfaces (AVI '06)*. ACM, 51–58. https://doi.org/10.1145/1133265.1133275

[9] Stéphane Chatty. 1994. Extending a Graphical Toolkit for Two-handed Interaction. In *Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology (UIST '94)*. ACM, New York, NY, USA, 195–204. https://doi.org/10.1145/192426.192500

[10] Stéphane Chatty. 2012. A unified framework for control structures in interactive software. (Sept. 2012). https://hal.archives-ouvertes.fr/hal-01800741 . Working paper. First version authored in 2012, archived in HAL in 2018.

[11] Stéphane Chatty, Mathieu Magnaudet, and Daniel Prun. 2015. Verification of properties of interactive components from their executable code. In *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. 276–285.

[12] Stéphane Chatty, Stéphane Sire, Jean-Luc Vinot, Patrick Lecoanet, Alexandre Lemort, and Christophe Mertz. 2004. Revisiting Visual Interface Programming: Creating GUI Tools for Designers and Programmers. In *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology (UIST '04)*. ACM, New York, NY, USA, 267–276. https://doi.org/10.1145/1029632.1029678

[13] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. 2005. A conservative extension of synchronous data-flow with state machines. In *Proc. of ACM EMSOFT'05*. ACM, 173–182. https://doi.org/10.1145/1086228.1086261

[14] B. Collignon, J. Vanderdonckt, and G. Calvary. 2008. Model-Driven Engineering of Multi-target Plastic User Interfaces. In *Fourth International Conference on Autonomic and Autonomous Systems (ICAS'08)*. 7–14. https://doi.org/10.1109/ICAS.2008.37

[15] Stéphane Conversy. 2014. Unifying Textual and Visual: a Theoretical Account of the Visual Perception of Programming Languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Splash Onward! '14)*. ACM Press, New York, NY, USA, 201–212. https://doi.org/10.1145/2661136.2661138

[16] Antony Courtney. 2001. Frappé: Functional Reactive Programming in Java. In *Practical Aspects of Declarative Languages: Third International Symposium, PADL 2001 Las Vegas, Nevada, March 11–12, 2001 Proceedings*, I. V. Ramakrishnan (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 29–44. https://doi.org/10.1007/3-540-45241-9_3

[17] Pierre Dragicevic and Jean-Daniel Fekete. 2004. Support for Input Adaptability in the ICon Toolkit. In *Proceedings of the Sixth International Conference on Multimodal Interfaces (ICMI'04)*. ACM Press, 212–219.

[18] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *International Conference on Functional Programming*. 263–273. http://conal.net/papers/icfp97/

[19] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The Synchronous Data Flow Programming Language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320.

[20] D. Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8, 3 (June 1987), 231–274.

[21] H. Rex Hartson and Deborah Hix. 1989. Human-Computer Interface Development: Concepts and Systems for its Management. *Comput. Surveys* 21 (1989), 5–92.

[22] C. A. R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21, 8 (1978), 666–677.

[23] Christophe Hurter and Stéphane Conversy. 2008. Towards characterizing visualization. In *Proceedings of the 15th conference on Design Specification and Verification of Interactive Systems (DSVIS 2008), Lecture Notes in Computer Science*. Springer Verlag, 287–293. https://doi.org/10.1007/978-3-540-70569-7_26

[24] ISO 9241-210:2010 2010. *Ergonomics of human-system interaction – Part 210: Human-centred design for interactive systems*. Standard. International Organization for Standardization, Geneva, CH.

[25] R.J.K. Jacob, L. Deligiannidis, and S. Morrison. 1999. A Software Model and Specification Language for Non-WIMP User Interfaces. *ACM Transactions on Computer-Human Interaction* 6, 1 (1999), 1–46.

[26] R. J. K. Jacob. 1983. Using Formal Specifications in the Design of a Human-Computer Interface. *Commun. ACM* 26 (1983), 259–264.

[27] Alan Jeffrey. 2013. Causality for Free!: Parametricity Implies Causality for Functional Reactive Programs. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification (PLPV '13)*. ACM, New York, NY, USA, 57–68. https://doi.org/10.1145/2428116.2428127

[28] Ralph E. Johnson and Brian Foote. 1988. Designing Reusable Classes. *Object-Oriented Programming* 1, 2 (1988).

[29] Alan C. Kay. 1996. History of Programming languages—II. ACM, New York, NY, USA, Chapter The Early History of Smalltalk, 511–598. https://doi.org/10.1145/234286.1057828

[30] Eric Lecolinet. 2003. A molecular architecture for creating advanced GUIs. In *Proceedings of the ACM UIST*. 135–144.

[31] Mathieu Magnaudet and Stéphane Chatty. 2014. What Should Adaptivity Mean to Interactive Software Programmers?. In *Proceedings of the 2014 ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '14)*. ACM, New York, NY, USA, 13–22. https://doi.org/10.1145/2607023.2607028

[32] Ingo Maier, Tiark Rompf, and Martin Odersky. 2012. *Deprecating the Observer Pattern*. Technical Report. EPFL.

[33] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: A Programming Language for Ajax Applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 1–20. https://doi.org/10.1145/1640089.1640091

[34] Robin Milner. 1980. *A Calculus of Communicating Systems*. Springer-Verlag.

[35] Brad Myers. 1991. Separating application code from toolkits: Eliminating the spaghetti of callbacks. In *Proceedings of the ACM UIST*. Addison-Wesley, 211–220.

[36] Brad Myers, Sun Young Park, Yoko Nakano, Greg Mueller, and Andrew Ko. 2008. How Designers Design and Program Interactive Behaviors. In *Proceedings of IEEE VLHCC'08*. IEEE Computer Society, 177–184. https://doi.org/10.1109/VLHCC.2008.4639081

[37] Brad A. Myers. 1990. A new model for handling input. *ACM Transactions on Office Information Systems* (July 1990), 289–320.

[38] Brad A. Myers et al. 1990. Garnet, Comprehensive Support for Graphical, Highly Interactive User Interfaces. *IEEE Computer* (Nov. 1990), 71–85.

[39] David Navarre, Philippe Palanque, Jean-Francois Jean-François Ladry, and Eric Barboni. 2009. 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. https://doi.org/10.1145/1614390.1614393

[40] G. E. Pfaff (Ed.). 1985. *User Interface Management Systems*. Springer-Verlag.

[41] Trygve M. H. Reenskaug. 1979. Models - Views - Controllers. (December 1979). http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf.

[42] Nicholas Rescher. 2000. *Process Philosophy: A Survey of Basic Issues*. University of Pittsburgh Press, Pittsburgh.

[43] Wesley Salmon. 1984. *Scientific Explanation and the Causal Structure of the World*. Princeton University Press.

[44] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications. In *Proceedings of the 13th International Conference on Modularity (MODULARITY '14)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/2577080.2577083

[45] Johanna Seibt. 2009. Forms of emergent interaction in General Process Theory. *Synthese* 166 (2009), 479–512.

[46] Johanna Seibt. 2013. Process Philosophy. In *The Stanford Encyclopedia of Philosophy*, Edward N. Zalta (Ed.).

[47] The Coq Development Team. 2018. The Coq Proof Assistant, version 8.8.0. (April 2018). https://doi.org/10.5281/zenodo.1219885

[48] Sam Van den Vonder, Joeri De Koster, Florian Myter, and Wolfgang De Meuter. 2017. Tackling the Awkward Squad for Reactive Programming: The Actor-reactor Model. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS 2017)*. ACM, New York, NY, USA, 27–33. https://doi.org/10.1145/3141858.3141863

[49] Atze van der Ploeg. 2013. Monadic Functional Reactive Programming. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell (Haskell '13)*. ACM, New York, NY, USA, 117–128. https://doi.org/10.1145/2503778.2503783

[50] Peter Van Roy. 2009. Programming Paradigms for Dummies: What Every Programmer Should Know. In *New Computational Paradigms for Computer Music*, G. Assayag and A. Gerzso (Eds.). IRCAM/Delatour.

[51] Peter Wegner. 1997. Why Interaction Is More Powerful Than Algorithms. *Commun. ACM* 40, 5 (1997), 80–91.

A APPENDIX

A.1 Volta power computation

```
Int current(-42)
Int current_left(-42)
Int current_right(-42)
Int vpack(-42)
Int power_kW(-42)

current_left + current_right => current
vpack * current / 1000 => power_kW
```

A.2 Volta low RPM alarm

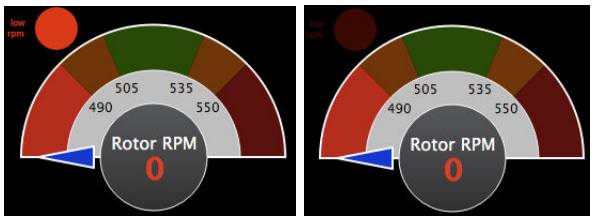


Fig. 9. The two graphical states of the blinking "low RPM" alarm

```
// in model.sma

Int rotor_rpm
Int rpm_motor_low_g(505)
Bool low_rpm(1)

rotor_rpm < rpm_motor_low_g => low_rpm

// in hmi.sma
Bool low_rpm (1)

Double dark_red (70)
Double luminous_red (255)

FillColor current_color (70, 0, 0)
Ellipse light_bulb (50, 10, 20, 20)
Text low (7, 8, "low")
Text rpm (4, 18, "rpm")

Switch onoff (true) {
  Component true {
    Clock clock (300)
    FSM fsm {
      State blinkup {
         luminous_red =: current_color.r
      }
      State blinkdown {
         dark_red =: current_color.r
      }
      blinkup->blinkdown (clock.tick)
      blinkdown->blinkup (clock.tick)
```

```
      }
    }

    Component false {
      dark_red =: current_color.r
    }
  }

  low_rpm => onoff.state

  // in model_hmi_connection.sma

  model.rotor_rpm => hmi.low_rpm
```

A.3 Volta current and power

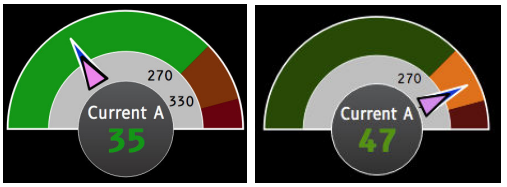


Fig. 10

```
_define_
rag_status(Component rect,
          Component cst, Component thr) {
   Int input (-1)

   Switch rag(undef) {
      Component red {
         cst.redc.r =: rect.fill.r
         cst.redc.g =: rect.fill.g
         cst.redc.b =: rect.fill.b
      }
      Component amber {
         cst.amberc.r =: rect.fill.r
         cst.amberc.g =: rect.fill.g
         cst.amberc.b =: rect.fill.b
      }
      Component green {
         cst.greenc.r =: rect.fill.r
         cst.greenc.g =: rect.fill.g
         cst.greenc.b =: rect.fill.b
      }
      Component undef {
         cst.undefc.r =: rect.fill.r
         cst.undefc.g =: rect.fill.g
         cst.undefc.b =: rect.fill.b
      }
   }
}

input < thr.low_a
```

```
    ? cst.reds
    : (input < thr.low_g
       ? cst.ambers
       : (input < thr.high_g
          ? cst.greens
          : (input < thr.high_a
             ? cst.ambers
             : cst.reds)))
     => rag.state
}

Component rag_thresholds_power_kW {
  Int low_a(-2) // unused, power can't be negative
  Int low_g(-1) // unused, power can't be negative
  Int high_g(70)
  Int high_a(90)
}

current = find(img.gauges.current)

addChildrenTo current {
   Int input_gauge(-1)
   input_gauge * 180 / 360
      => current.knob.needle.rotate.a

   Int input_text(-1)
   rag_status
      rag_current_value(current.rag_status.value,
        rag_constants, rag_thresholds_power_kW)

   input_text => rag_current_value.input
   input_text => current.rag_status.value.text
}
```

A.4 Volta HMI page management

```
page = find(img.lower_panel.right.page)
addChildrenTo page {
    Component page_up
    Component page_down
    Switch status_control(page1) {
       page1 << svg.page_def.page1
       page2 << svg.page_def.page2
       page3 << svg.page_def.page3
    }
    FSM sc_fsm {
       State page2
       State page3
       State page1
       page1 -> page2 (page_up)
       page2 -> page3 (page_up)
       page3 -> page1 (page_up)
       page1 -> page3 (page_down)
       page2 -> page1 (page_down)
       page3 -> page2 (page_down)
    }
    sc_fsm.state => status_control.state
}
```



Fig. 11. The three pages of the Volta HMI, and the left-right switch on the joystick to navigate them. The switch is connected to two GPIOs on the embedded platform.

```
// page down / page up
// with non-visible rectangles as push-button
Component mouse_page_up_page_down {
    GNoOutline noo
    GFillColor black(0,0,0)
    GTranslation t(340, 265)
    GUIRectangle page_down(0, -5, 60, 20, 5, 5)
    GUIRectangle page_up(60, -5, 60, 20, 5, 5)
    page_down.press -> page.page_down
    page_up.press -> page.page_up
}

// page down / page up
// with left-right switch through GPIO
Component page_gpio {
    gpio_down = find(gpio:B4.in)
    gpio_up = find(gpio:B23.in)
    gpio_down -> page.page_down
    gpio_up -> page.page_up
}
```

A.5 Primary Flight Displays

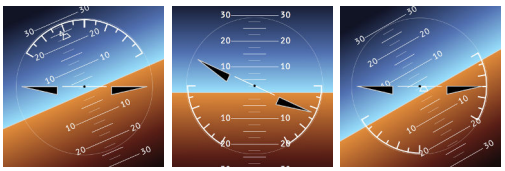


Fig. 12. The occidental, russian and new PFD

```
Double zero (0)
Double one (1)
Int radius (150)

FSM pfd_mode {
   State occidental {
      zero =: pfd.ATT.aeroplane_bank
      -aeroplane_bank => pfd.ATT.horizon_bank
      -aeroplane_bank => pfd.ATT.pointer_bank
      zero =: pfd.ATT.bankscale_bank
      -aeroplane_bank => pfd.ATT.pitchscale_bank
      -radius =: pfd.ATT.slip_trans_y
      one =: pfd.ATT.pointer_opacity
      one =: pfd.ATT.slip_opacity
   }
   State russian {
      aeroplane_bank => pfd.ATT.aeroplane_bank
      zero =: pfd.ATT.horizon_bank
      zero =: pfd.ATT.pointer_bank
      zero =: pfd.ATT.bankscale_bank
      zero =: pfd.ATT.pitchscale_bank
      //-10+zero =: pfd.ATT.slip_trans_y // agnostic
      zero =: pfd.ATT.pointer_opacity
      zero =: pfd.ATT.slip_opacity
   }
   State newpfd {
      zero =: pfd.ATT.aeroplane_bank
      -aeroplane_bank => pfd.ATT.horizon_bank
      zero =: pfd.ATT.pointer_bank
      -aeroplane_bank => pfd.ATT.bankscale_bank
      -aeroplane_bank => pfd.ATT.pitchscale_bank
      -10+zero =: pfd.ATT.slip_trans_y
      zero =: pfd.ATT.pointer_opacity
      one  =: pfd.ATT.slip_opacity
   }

   russian->occidental (controller.occidental.press)
   newpfd->occidental (controller.occidental.press)

   occidental->russian (controller.russian.press)
   newpfd->russian (controller.russian.press)

   russian->newpfd (controller.new.press)
   occidental->newpfd (controller.new.press)
}
```