Designing, developing and verifying interactive components iteratively with djnn

Stéphane Chatty, Mathieu Magnaudet, Daniel Prun, Stéphane Conversy, Stéphanie Rey, Mathieu Poirier

▶ **To cite this version:**

Stéphane Chatty, Mathieu Magnaudet, Daniel Prun, Stéphane Conversy, Stéphanie Rey, et al.. Designing, developing and verifying interactive components iteratively with djnn. 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016), Jan 2016, Toulouse, France. hal-01292291

HAL Id: hal-01292291
https://hal.science/hal-01292291

Submitted on 22 Mar 2016

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.

Designing, developing and verifying interactive components iteratively with djnn

Stéphane Chatty Mathieu Magnaudet Daniel Prun
Stéphane Conversy Stéphanie Rey Mathieu Poirier
Université de Toulouse - ENAC
7 av. Edouard Belin, 31055 Toulouse, France
{firstname.lastname}@enac.fr

ABSTRACT

Introducing iterative user interface design methods into the development processes of safety-critical software creates technical and methodological challenges. This article describes a new programming paradigm aimed at addressing some of these challenges: interaction-oriented programming. In this paradigm any piece of software consists of a hierarchical collection of components that can interact among themselves and with their environment, and its execution consists in propagating activation through interactions between components. We first describe the principles of interaction-oriented programming, and illustrate them by describing the basic components provided by the djnn programming framework to create interactive software. We then show how interactive programming provides a basis for formulating and checking properties that capture requirements on interactive components. The rest of the article is dedicated to example design and development scenarios that illustrate how development environments could leverage interactive programming in the future so as to jointly address the requirements of modern user interface design and safety-critical software development.

ACM Classification Keywords: H5.2 Information Interfaces and presentation: User Interfaces; D2.2 Software Engineering: Design Tools and Techniques; D3.3 Programming Languages: Language Constructs and Features.

Author keywords: interactive software, design process, development process, interactive component, software verification

INTRODUCTION

There are well established methods for developing safety critical software, such as those prescribed in the DO178 standards. In some domains, solutions have been developed to apply these methods to user interfaces as well. For instance, in aeronautics the ARINC 661 standard defines a collection of well known interactive components (or widgets), such as lists and menus. The protocol of these components is defined in such a way that they can be developed and certified individually, then reused at will. Industrial tools have even been introduced to assemble them graphically, then generate the corresponding code.

However, with ongoing plans to introduce more modern user interfaces in these safety critical settings, new questions appear. Not only are the envisaged interactive components often more complex and more difficult to specify than their predecessors, there is also a trend toward interface customization. Each aircraft or car manufacturer wants to have its own distinctive signature, in terms of interaction and not only graphical appearance. Consequently, they expect equipment providers to deliver products whose behavior and appearance can be modified. This means that one cannot rely solely on industry standards that define interactive components, and that methods must be proposed for designing and developing custom components, usually as a collaboration between an equipement provider and an integrator.

The user interface industry has developed and validated methods for designing custom interactive software, and recent R&T projects have shown that they can be applied successfully by the aeronautics industry. However these methods involve various actors, including human factors experts and designers, and are iterative by nature. Integrating them in industrial development processes brings new challenges, and appropriate tool chains are not yet available to support this. In addition, the state of the art in interactive software development has not yet reached the same level of maturity as other branches of computer science. Interactive software is still complex and costly to produce [18, 19], and model driven engineering methods are not yet widely used. This does not provide a favorable context for developing and validating custom components on a repeated basis. In particular, validating interactive software that contains code written in a traditional programming language is very costly.

In this article, we propose a theoretical and practical contribution towards the resolution of these two issues: the convergence of user interface design methods and critical software development processes, and the validation of custom interactive software. This contribution consists of a software execution model and a software architecture, whose combination provides the basis of what we call interaction-oriented programming. This allows to organize interactive software as a collection of components whose execution can be analyzed and whose definition can be incrementally refined. The proposed execution model and component architecture are implemented in a development framework named `djnn`, that has been used successfully in various projects.

We first analyze reasons why it is important to account for interactive software design methods. Doing so, we outline requirements for future design processes and tools that would combine the needs of interactive and safety critical systems. We then stress the key role of software architecture and execution models in fulfilling these requirements. After reviewing the state of the art on these topics, we introduce a theoretical framework for interactive components that defines a software architecture and an execution model. We then describe the djnn framework and use a simple aeronautical component (a primary flight display) and a series of idealized development scenarios to illustrate how new processes and tools for designing interactive software can be derived from this work.

ANALYSIS: WHY IS INTERACTIVE SOFTWARE SPECIAL?

Managing hidden requirements

One of the keys to successfully developing complex systems is the management of requirements. Gathering requirements, analyzing them, and tracing them in products represent an important part of the development effort. Various development methods and tools have been proposed for this purpose, and for supporting certification processes. Unfortunately, user interfaces suffer from a fatal flaw with this regard: requirements are impossible to gather in advance. There are hidden requirements that keep appearing during design and development processes, or even during the use of the final systems. As an example, consider an equipment in which two critical information fields are shown side by side. The two graphical representations may work perfectly when tested independently, and when using them together visual interferences or perceived inconsistencies between them can lead users to errors in reading them. It can also be discovered later that one of these representations does not work well when users are in particular mental conditions that had not been anticipated. Some of these emerging requirements, when not caught early enough, can be palliated by operational procedures or user training. Others can lead to safety flaws, or to outright rejection of the system by its intended users.

A day might come when cognitive sciences provide us with enough knowledge that requirements can actually be gathered in advance, but this is at best a long time goal. Until then, the best known solution for securing requirements is iterative design, a flavor of agile methods developed specifically for interactive software. Users are presented prototypes created by designers and usability experts, new requirements emerge from the confrontation, and new prototypes can be designed. During this process, two collections are incrementally created: a collection of design elements, and a collection of requirements. It is only after a number of iterations that the two collections can be considered sufficient.

This iterative process is not intrinsically incompatible with the processes used in safety-critical developments. Nevertheless, not only does it require dedicated tools for the initial phases of design, it also interferes with further development phases. In theory, user interface design could be done prior to actual software development, so that the two processes are independent. In practice however, design generally continues in parallel with development, if only because hidden requirements keep showing up. This means that solutions must be proposed to manage how the two processes interact with each other.

Architecture of interactive software

The above provides a first set of requirements for tools dedicated to user interface design in critical systems: an architecture that allows the incremental refinement of design elements and their execution, as well as the incremental definition of requirements and their checking. The desire to create customizable user interfaces highlights additional requirements: the ability to separately define the behaviors of given components, and check that the resulting user interfaces still meets the requirements.

Interoperability and interchangeability of software components is actually a universal concern, and it is the essence of software architecture. There are various situations in which programmers need to change how components are combined in their software. This includes early design phases, in which prototypes are developed. And this also includes various refactoring phases, whether during initial development or when the software is reused to create new products. Each programming paradigm provides support for the interoperability of a certain type of components, and therefore for software architecture. For instance, functional programming makes it easier to refactor software that performs computations. Similarly process algebra makes it easier to refactor software that reacts to input.

However, interactive software brings its own class of refactoring. There are many different ways to design a user interface for a given task. Take for instance the single pressing of a button to activate a function. The button can just change state visually, for instance by changing color. This atomic change can be expressed through an assignment, either of the color itself or of a symbolic state of the button. Alternatively, the button can change with an animation. This would require the state assignment to be replaced with a process that repeatedly modifies the visual state, and that can continues in parallel with further interactions. These two options are profoundly different in existing programming paradigms, and this makes refactoring costly. To compound matters further, user interfaces can make use of various interaction modalities such as graphics, sound, touch input, speech input, eye tracking, etc. Not only is each modality different from the others, they can also be combined in ways that require to manage interaction state, time intervals, and other complex execution patterns.

Various architecture patterns have been proposed to support the interoperability of components in interactive software. As exemplified above, some situations are easily expressed by simple constructs in existing programming languages, such as function calls, assignments or loops. Others require specific solutions such as events, data-flows, state machines. This diversity of control structures and component interconnection mechanisms, if not derived from a reduced set of primitives, hampers interoperability. It also limits the formulation of an execution semantics for programs, and therefore the ability to formulate and check properties on software components.

Another consequence is in the complexity for programmers. Each individual solution alleviates one source of complexity, for instance state machines make it easy to manage state dependencies. But as soon as several solutions are mixed, new sources of complexity appear. For instance, languages such are QML must be combined with traditional languages such as C++ to produce full applications, thus reintroducing in programs the traditional architecture patterns and control structures. Similarly, combining state machines with dataflows can rarely be performed without writing additional code in a traditional programming language. Programmers therefore end up manipulating independent concepts that each describe part of the program execution, and that do not have a clear common base.

Therefore, defining a minimal set of primitives from which the behavior of interactive software can be derived is required both for raising interactive software to the same level of manageability as other types of software, and for supporting development processes that rely on software refactoring.

STATE OF THE ART

Complexity

Various methods have been proposed over the last decades to make interactive software development less complex. The initial approach, mostly in the 1980s and 1990s, consisted in creating User Interface Management Systems and user interface toolkits on top of traditional programming languages. This approach evolved into two areas of research. On the one hand were collections of reusable interactive components such as dialogue boxes and buttons, that evolved into specialized languages and standards such as XAML, XUL, QML and ARINC 661.

On the other hand were software patterns aimed at managing the architecture requirements of interactive software, that differ significantly from those of computation-oriented software. Some of these patterns were aimed at separating the interactive code from the computation-oriented and data-oriented code. This includes the Seeheim and ARCH architecture, and the MVC, PAC and MVVM patterns. Other patterns were aimed at providing better support for the execution and control patterns encountered in interactive software. This starts with callback functions and the Inversion of Control pattern that account for the prevalence of external control in interactive software. But this also includes more complex patterns to support state management (state machines, hierarchical state machines, Statecharts) and dataflow (one-way constraints, functional reactive programming [11], dataflow bricks [6]). However, as mentioned previously, these solutions provide only local complexity relief: they make some parts of the behavior easier to formulate, but the overall complexity remains high because of how heterogeneous constructs are combined.

Development process

Historically, two approaches have been taken to develop graphical user interfaces. The first consists in giving a functional specification to programmers, and rely on their graphical skills. The second, used when visual quality is desirable, consists in asking graphical designers to produce visuals then asking programmers to reproduce them in their programs. Having programmers recode the graphics like this is both a waste of time and a cause of errors.

In contrast with this, traditional programmers can split their tasks and work in parallel, relying on well supported integration techniques such as separate compilation and linking. Interactive software developers could take advantage of similar solutions. First, they could to split their tasks and work in parallel with graphical designers. Then, they could also split tasks during iterative design phases with users and human factors specialists. Solutions have been proposed for this, leveraging on the similarities between the visual and logical structure of a user interface and the abstract syntax tree in traditional programming [7]. Considering graphics and interactive behaviors as nodes of a tree allows to produce them independently, then build the tree during a loading phase at compilation or execution time, and execute the resulting tree. This has been shown to yield significant improvements both in terms of effort and development time span[**?**]. However, this is only feasible for those parts of the software that can be consistently modeled as nodes in the execution tree. To be fully operational, this approach requires that 1OO% of programs can be modeled in the same framework.

Another approach consists in applying model-driven engineering methods to interactive software. In this approach, domain experts produce an abstract model of the task to be carried out by users, then all or part of the software is derived from this abstract model. However this only work for some classes of user activities, and the resulting user interfaces are stereotyped and of limited usability. More practical variants of this approach have also been tested, where the user interface is expressed using a theoretical model, but it is produced through a design process rather than generated from a task model. This allows to take advantage of the benefits of model-driven architectures without losing the interface quality brought by the design process. For instance in the PetShop system, applications are created by combining Object Petri Net models with Java code [20]. However, like above this approach will reach its full potential only when 100% of the user interface can be expressed in the model.

Code verification

During the last decades, various methods dedicated to the verification of interactive system properties have been proposed. The widely used approach relies on the test of the final system (or a prototype of it) where end-users accomplish selected tasks in a dedicated environment. Observations and measurements performed during the execution are used to assess whether the system fulfills the expected properties or not. The main drawback of this approach lies in its lack of exhaustivity because properties cannot be checked against all possible executions of the system. Moreover, this approach can only be used after the development (the system must be available) which constitutes another negative point: bugs are way more expensive to locate and correct at this development stage.

Model-based methods have been proposed to minimize these issues: at design time, a model of the future system is built

and properties are verified by studies performed on the model rather than on the final system itself. The underlying logic behind this approach is that if a property holds on the model, and if the final system is built according to the model, then the property holds on the final system. Model checking aims at verifying properties on the state-transition structure built during the simulation of the model [20]. This is very similar to methods used for safety-critical software (eg. Esterel: [4], Scade: [12]). Alternatively, proof-based methods consist in mathematically proving the preservation of properties (invariants, pre-conditions and / or post-conditions) during successive refinements of the model (VDM: [10], Z: [16] or B: [2]).

Although they have been shown to be very effective, such model-checking approaches suffer from their impossibility to encompass systems with infinite number of states or transitions which is a major limitation in the context of interactive systems. Moreover, the assumption that "the final system is built according to the model" is hard to reach for reasons explained previously: no available model describes 100% of a user interface. To palliate this, some authors introduce a simplified model called the abstract user interface, in which the totality of a user interface can be described. Properties can be checked on this model. However, the process of converting this into a concrete user interface by adding code to the abstract user interface limits the benefits that can be obtained from the model based approach.

Abstract interpretation ([9]) is a verification method based on static analysis of the software code. An abstract semantic is extracted from the code and can be used to verify properties and perform optimisations [1]. This approach has historically been used for the verification of various properties of programs. For instance, the static analyzer Astrée is able to prove the absence of some types of run time errors on C programs ([5]), and has been used in safety-critical projects. For interactive properties, [17] first proposed this approach with the objective of providing a unification canvas for verification techniques. [14] described the verification of interactive Web pages through the static analysis of the user interface with ergonomic rules, encoded in UsiXML. [21] proposed to build and exploit a graph-oriented semantics of an interactive device to support the verification of properties. In this work, an existing device was analyzed and modeled with a graph whose arcs represent user actions and nodes observable states. It was then possible to compute some interactive properties on the graph that can be interpreted at the device level.

Abstract interpretation is an efficient application of the model checking approach: work is performed directly at the lowest level, that is the code of a concrete user interface, and therefore avoids the limitations of using an abstract model. Paradoxically, its main drawback for interactive systems lies in the impossibility to access to higher levels of description of the interface, such as those contained in the abstract user interface, because they usually have disappeared during implementation and compilation. It becomes impossible to check properties that would be expressed at these higher levels, such as "is this rectangle red when this button is pressed?". When the software is built with languages that do not capture the appropriate level of information (e.g. C ot Java), the missing information must be introduced by producing a model by hand.

[1]Concrete semantics are mathematically well defined objects that explicit the meaning and the possible behaviors of the program. Concrete semantics are generally not computable, which makes all non-trivial properties undecidable. To avoid this, abstract semantics are introduced as computable approximations of concrete semantics in which more properties are decidable.

Analysis and positioning

Most limitations of the state of the art derive from the same cause: the limited power of expression of the patterns, languages and models available. Each of the available solutions for expressing parts of interactive software have focused on a given requirement that was not fulfilled by traditional programming languages: supporting external control, or providing reusable dialogue boxes, or managing state, etc. Each alleviates one source of complexity in interactive software, but none provides a complete solution like modern programming languages do for computation-oriented software. This has consequences on software complexity, on development processes, and on the ability to verify code properties.

In this article, we describe an alternative solution that overcomes these limitations using an approach similar to Lustre or Esterel [15, 4]: adopting an execution model dedicated to the category of software that is being developed, and using it to develop the totality of the concerned software. Like Lustre and Esterel, the proposed model represents concurrency with the concept of process; it can be seen as a third point of view on interactive systems, more adapted to the need of user interfaces and making interaction its core concept. In contrast with them, its design integrates software engineering concerns and particularly the need of strong conceptual unification so as to support flexible development processes. Moreover, our work extends the approach followed by SCADE Display concerning the verification capabilities of interactive applications: when SCADE adresses low-level related properties (related to the concrete user interface: graphics, code design) our approach also encompasses upstream user interface design phases.

THE DJNN FRAMEWORK

`djnn` (available at `http://djnn.net`) is a programming framework that relies on a model of interactive software in which any program can be described as a tree of interactive components [7]. Basic components such as variables, control structures, and graphical objects are assembled to produce bigger components, themselves assembled until producing the desired application.

The execution of a program is described by the interactions between its components, and between them and the external environment: components react to events detected in their environment, and may themselves trigger events. For instance, a simple "fire alarm" program can be described with three components. The first is activated when the temperature is higher than a threshold, the second produces a sound when

activated and the third binds the two others by propagating the activation of the first one to the second one.

Such a component model applies to input and output devices. This allows djnn to provide support for a wide range of devices, thus fostering the exploration of wide design spaces. But djnn has also the expressive potential of a general programming language. This contrasts with most user interface programming frameworks, which provide reusable components and architecture patterns that programmers combine with code written in a traditional programming language. Not only does djnn aim at covering 100% of the user interface code, it also has the potential of describing the functional core as well, thus covering whole interactive applications.

Theoretical foundation: interactive processes

Like functional programming languages, the conceptual model of djnn relies on a very reduced set of basic concepts from which all other language concepts and programmer-defined concepts are derived. In functional languages the basic concepts are functions, arguments and function calls, all rooted in the theoretical concept of lambda term from lambda calculus. In djnn the basic concepts are components, names and activation, all rooted in the theoretical concept of process from process algebras [3].

While computations can be described as the evaluation of lambda expressions, interactions can be described as the activation of interconnected processes: activation signals, called events, propagate from one process to another according to how they are coupled, thus producing the reactive behavior of the system. This concept of interaction differs from that of communication used in most process algebras, but it is nevertheless possible to express formal models of component activation in existing process calculi.

Process-based theories are general enough to model both interaction-oriented software and computation-oriented software [13]. But they can also model hardware devices, and more generally the environment in which software applications run. This allows to model both the software and its direct environment using processes, so that no special provision has to be made for those part of the software that interact with the environment and the user. Whole interactive applications can therefore be described in the same language.

Core interactive components

Components are man-made embodiment of processes: engineers build systems by assembling components, and the theoretical model of the resulting systems can be deduced from those of the individual components and how they are assembled. Programmers create interactive programs by instantiating and assembling software components, and connecting them to hardware components. The djnn environment provides them with basic software components to this effect:

- input components represent input devices, sensors, or elements of the execution context. Their activation is coupled to external events. For instance, a mouse is an input component made of smaller input components such as a buttons and a position tracker. The position tracker itself has two smaller input components named X and Y.
- output components represent output devices or abstractions used to manipulate output devices. Their activation is coupled to actions performed by the output devices. For example, a graphical object is an output component and any activation of it or of its sub-components triggers changes in the display.
- data components represent the computer memory. The smaller data components are named properties, and correspond to the basic types. Properties have sub-components named READ and WRITE, but like in traditional languages the usual practice is to reserve their use to specific components such as operations as assignments. The activation of the WRITE sub-component is the basis of dataflow: modifying a value can be used to trigger actions such as copying the value to another.
- operation components represent the operations provided by the execution platform, usually the CPU.
- control structure components represent the various ways in which the activation of components can be controlled. A control structure is a component that creates couplings between other components when it is activated. Control structures are the key to supporting the diversity of control patterns used in interactive software. Standard control structures range from the binding, which creates a simple coupling when it is activated, or the connector, which creates a coupling between a source and a copy instruction, to finite state machines and Statecharts. Other control structures can be created at will by combining existing components.

Assembling components

Complete applications and reusable components can be created from the basic components described above. Basic components are assembled to create large components that implement small interactors. These interactors can be assembled to create larger interactors, and so on.

A simple button, for example, can be built from few graphical shapes (rectangles, gradient, color, text, etc.) associated with a switch and a finite state machine that control the appearance of the button on mouse or touch events.

```
component button (posX, posY, label) {
  component click
  switch sw {
    component released {
      color c1 (200, 200, 200)
    }
    component pressed {
      color c2 (100, 100, 100)
     }
  }
  rectangle rec (posX, posY, 60, 30)
  text t (posX + 15, posY + 15,  label)
  FSM fsm {
    state released
    state pressed
    transition (released, press, rec, "press", 0)
    transition (press, released, rec, "release", click)
  }
  connector (fsm, "state", sw, "state")
}
```

The button can then be inserted in a more complex component to trigger a specific process through a simple binding.

```
component alarm {
  button b1 (50, 50, "alarm")
  beep fire_alarm
  binding (b1, "click", fire_alarm, "start")
}
```

Verifying properties

The djnn framework gives a central role to the tree structure of programs. In particular, mimicking graphical scene graphs introduced a few decades ago, the tree structure is used to express execution control. As a consequence it becomes possible to evaluate some properties by a static analysis of the tree through pattern matching techniques such as XPath requests. For example, the position of a graphical object in the tree tells its relative position to other graphical objects in the same component. Thus, a component situated on the right of another one in the tree will be displayed on top of it if their coordinates overlap. In the same way, djnn implements a flavor of graphical scene graphs in which graphical style components such as color, opacity and stroke width can be placed in the tree and act as context modifiers that affect all the shapes that follow see Figure 1).

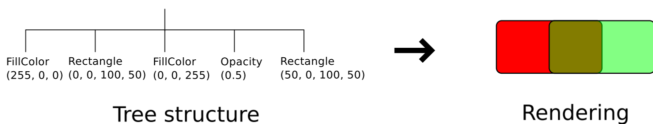

Figure 1. The order in the tree determines the graphical result

Pattern matching over the tree structure also enables the verification of component signature. This is a strong requirement in the context of the parallel development of complex graphical user interfaces where it is needed to ensure that a component respect a contract. Suppose for example that one creates a new widget for a WIMP toolkit. It must then be checked that this widget has a `width` and a `height` children to ensure that it will be possible to connect them to the layout system. Such a verification can be made by checking that the XPath expressions `expr=(/widget/width)` and `expr=(/widget/height)` over the component do not return a null result.

Finally, the combination of Xpath queries to select elements and simple algorithms over their results allows to address more complex case such as the verification of the control flow connecting a sensor to an alarm through various computation units [8].

The djnn platform

djnn is currently available as a collection of libraries and as an interpreter. The core djnn library manages the execution model and the component system. The other libraries each bring a collection of components dedicated to an interaction modality: graphics, input, gesture, sound, etc. Programmers can use these libraries like they would use any programming framework, by writing programs that use the djnn API to create components. The main difference with programming frameworks is that they can create entire applications with component-creation instructions only.

The core djnn library also implements parsers for external component formats, currently XML and Json. Therefore, it can be turned into a component interpretor that reads components from files and executes them. Components can be stored in multiple files in order to support various software engineering processes. In particular, user interface design teams like to store their graphical components in separate files, so that they can be managed independently by graphical designers and merged with the rest of the application at run time only.

APPLICATION SCENARIOS

As an example development process based on djnn, consider an aircraft manufacturer who wants to add new interaction functionalities on a particular cockpit subsystem. This translates into various engineering phases at growing technology readiness levels, from initial exploration to final system development and validation. At lower TRLs, the design processes will ideally reflect the state of the art of user interface design. At higher TRLs, they must reflect the state of the art of requirements engineering and code validation.

In this section, we illustrate a proof of concept experiment where, after observing current practices and gathering requirements from various actors of the aeronautical industry, we tested the role that djnn could play in supporting work at both lower and higher TRLs. The proposed scenarios are very simplified and exaggerated compared to the original industrial situations. Nevertheless, they serve well to illustrate the benefits that could be expected from a unified architecture and execution model such as djnn's.

Figure 2. A primary flight display

We focus here on the design process of a primary flight display (PFD) i.e. the instrument that displays the basic parameters of the plane. The PFD consists of six parts, as shown in Figure 2:

- Attitude indicator: also known as artificial horizon, it gives information about the pitch (fore and aft tilt) and roll (side to side tilt) of the aircraft (center of the PFD)
- Altitude indicator (right side).
- Airspeed indicator (left side).
- Heading display: displays the magnetic heading of the aircraft (bottom).
- Source selector buttons: control the display of radio magnetic indicators on the heading display (bottom).
- Alerts: alert messages show at the top of the display.

In this prototype, the attitude component has an additional interactive part. Besides displaying altitude of the plane, this component gives an indication on the target altitude for an autopilot. However, while in traditional cockpits, the pilot sets the target altitude with a physical button in a distant part of the cockpit, here the team wants to explore the direct manipulation on the PFD.

In the following scenarios, a group of designers work with test pilots to build a prototype of the graphical appearance and the behavior of a new PFD, then the prototype is sent to the equipment provider. There, a team of developer codes the component and runs automated checks to verify that it has the same behavior as the prototype, while another team runs checks to verify properties that the component needs for inclusion in the existing code-base. Later, the component is returned to the manufacturer, who can run automated checks to verify its conformance with the original prototype. Through the examination of the design process of two components, we show here how the `djnn` framework supports concurrent development and enables the verification and validation of components.

Increasing fidelity

In the cockpit, the PFD is part of a more complex system of sensors and physical interactors. The team needs to simulate these subsystems in order to test and refine the behavior of the attitude component. At low TRL, the sensors do not have to be realistic. Therefore, the developer chooses to connect the attitude indicator to the best compatible sensor at hand: the motion sensor of his laptop computer. This allows him to perform very fast development iterations. When he is satisfied with his work, the developer needs to send the component to the lead developer who is in charge of integrating several components. He dumps the component to XML format and sends the resulting file to her. The lead developer just has to drag and drop the XML file to her project directory, where it will sit with the other components she has received. Running the master PFD component will load all the components from the directory and execute them. She can use her own motion sensor for testing the result. When the component is mature enough, it can undergo a review process to migrate to a higher TRL. The team in charge of this must test the component against a more realistic environment: a simulation engine controlled by a joystick. They save the component to a directory on the test machine with the proper equipment. The component needs to be adapted to the joystick used for pitch and roll. If necessary, this can be done with a simple rewiring; no modification is required in the component itself. But here, the original developer has added an adaptive behavior to the component: when a joystick is detected, the rewiring is performed automatically, and the component can be used as is.

Concurrent development

A programmer and a graphical designer are producing a component that displays the heading of the plane. Using the initial specification of the component structure as a contract, they can work in parallel. While the designer is creating the graphical skin and layout of the component, the programmer implements the behavior. Here, she uses temporary graphics that the graphical designer has sent earlier (Figure 3a).

When the graphical designer produces graphics, he saves them in SVG format, an XML-based markup language supported by all major vector graphics authoring tools. In this form, the graphics are considered as `djnn` components and can be manipulated like any other component. The developer can add them to the component he is working on, address them by their name, and connect them to other subcomponents. The names of graphical components in the SVG file are the implementation of the contract between the graphical designer and the developer.

When the final graphics are ready (Figure 3b) the graphical designer can send them to the developer. Replacing the temporary graphics with the new ones is as simple as replacing the SVG file in the appropriate directory and restarting the component. In our case, the graphical designer is late and the component has already been sent to the project manager who gives a demo to visitors in a few minutes. This poses no problem: the project manager saves the file, restarts the component, and the demo is ready.

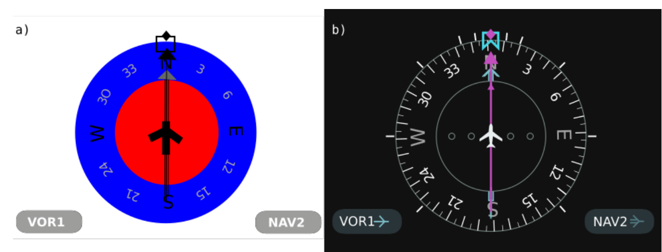

Figure 3. Heading display: (a) sketch (b) final graphics

Parallel design

The altitude component, besides displaying the altitude of the plane, also gives an indication on the target altitude for an autopilot. In traditional cockpits, the pilot sets the target altitude with a physical button in a distant part of the cockpit. For this new design, the team wants to explore the direct manipulation on the PFD. During design sessions, the team has identified several interaction variants. One prototype is developed in order to explore them further with final users. The first option

relies on sliding the element that represents the target altitude (the blue element in Figure 4a). The second option is an indirect interaction on a smartphone-like number picker (Figure 4b). Switching from one option to the other is just a matter of loading one component or the other. Here, usability tests highlight manipulation problems with the first option and pilots validate the second one.

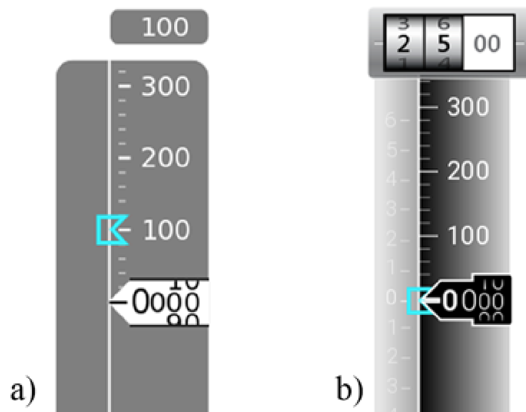

Figure 4. Altitude indicators: (a) direct manipulation and (b) indirect interaction for setting the target altitude

Component verification and validation

The airspeed component is subcontracted to an external provider, who delivers it through a web site. The project manager can either download the component and copy it in the component directory as described above or have it loaded dynamically by referencing the URL in the program. He needs to verify that the component will work well with the rest of the application. To do so he runs an automatic verification against the specification of the component, made of a collection of properties. This verification is made on the XML form of the `djnn` component, using pattern-matching techniques. In the first version of the component, the "ground speed" property is missing and the verification process automatically notifies it. After notifying the subcontractor, the verification is successful for the revised version received, and the component is loaded.

After the integration of all sub-components to the PFD component, the lead developer verifies that all the control flows are well wired. For example, she needs to be sure that alarms will be displayed when required. She uses a program that checks the control flow. The program traces back the control chain from "alert terrain" up to the "altitude" property and signals that it is not connected to any value. In the mean time, the verification raises a warning on the attitude component: two dataflow chains are connected to the same value. Thanks to these warnings, she can deduce than the simulated value of the altitude is connected by mistake to the attitude property, correct it and be certain that the alert terrain alarm will be displayed when needed.

CONCLUSION AND PERSPECTIVES

In this article, we proposed a general framework that provides a semantical unification of control mechanisms in interactive software. This framework relies on a framework whose basic elements are processes, names and events, from which more complex control structures can be defined (transfer of control, activation, interruption, transfer of data, deterministic choice, state machines, etc). On the top of these, a large collection of components have been designed for implementing the pragmatic aspects interaction: management of display (graphics, text, color, ...), input management (mouse, multi)touch), sound, interface with various external devices, etc. `djnn` is the name of the proposed implementation of this framework. It provides interaction designers and software developers with a platform for developing new components (through a dedicated language), assembling them and running the resulting application.

One long term objective for `djnn` is to provide the industry with a tool for development of interactive applications adapted to their domains. For this purpose, two major axes must be studied: for the moment, `djnn` is not supported by an IDE (Integrated Development Environment) similar to Codeblock, Eclipse or TopCased. Such tools useful for `djnn` users must be developed, especially a graphical editor for `djnn` models. The question of certification also has to be tackled. In the context of aeronautic, if `djnn` is used to develop and verify aircraft on-board applications, it must comply with some normative requirements related to software tool qualification (as specified in [1]).

The framework encompasses mechanisms dedicated to the expression and the verification of properties specific to interactive applications. Based on abstract interpretation, this allows to directly check on the code various properties at design- or system-level. Abstractions based on the component graph as well as the control flow graph retrieved from the code allow to address properties related not only to low level abstraction (code) but also to higher level (user interaction). As all necessary information describing the interaction is available at code level, our approach does not suffer from classical impossibilities related to absence of high level information for verification. Even if promising results for verification have already been obtained, a lot of research remain to be done to explore all the possibilities enabled by this approach.

ACKNOWLEDGEMENTS

This work was supported by DGAC projects Fenics and Fumseck, and the EU ARTEMIS JU through project HoliDes (http://www.holides.eu/) SP-8, GA No.: 332933. Any contents herein reflect only the authors' views. No supporting agency is liable for any use that may be made of the information contained herein.

REFERENCES

1. DO330. Software Tool Qualification Considerations, 2012. RTCA,Inc.

2. Aït Ameur, Y., Baron, M., Kamel, N., and Mota, J.-M. Encoding a process algebra using the Event B method. *STTT 11*, 3 (2009), 239–253.

3. Baeten, J. C. M. A brief history of process algebra. *Theor. Comput. Sci. 335*, 2-3 (May 2005), 131–146.
4. Berry, G., Bouali, A., Fornari, X., Ledinot, E., Nassor, E., and De Simone, R. ESTEREL: A formal method applied to avionic software development. *Science of Computer Programming 36*, 1 (Dec. 2000), 5–25.
5. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Mine, A., Monniaux, D., and Rival, X. *Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software*. LNCS 2566. Springer, Oct. 2002, 85–108.
6. Chatty, S. Extending a graphical toolkit for two-handed interaction. In *Proceedings of the ACM UIST*, Addison-Wesley (Nov. 1994), 195–204.
7. Chatty, S. Supporting multidisciplinary software composition for interactive applications. In *Proc. of the 7th international symposium on software composition*, no. 4954 in LNCS, Springer Verlag (2008), 173–189.
8. Chatty, S., Magnaudet, M., and Prun, D. Verification of properties of interactive components from their executable code. In *Proc. ACM EICS'15*, ACM (2015).
9. Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. ACM POPL'77*, ACM (1977), 238–252.
10. Duke, D. J., and Harrison, M. D. Abstract Interaction Objects. *Computer Graphics Forum 12*, 3 (1993), 25–36.
11. Elliott, C., and Hudak, P. Functional reactive animation. In *International Conference on Functional Programming* (1997), 263–273.
12. Esterel Technologies. Scade display HMI software design. `http://www.esterel-technologies.com/products/scade-display/`.
13. Goldin, D., Smolka, S., and Wegner, P., Eds. *Interactive computation - the new paradigm*. Springer-Verlag, 2006.
14. González-Calleros, J. M., Guerrero Garcia, J., and Vanderdonckt, J. Advanced human-machine interface automatic evaluation. *Universal Access in the Information Society 12*, 4 (2013), 387–401.
15. Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. The synchronous dataflow programming language Lustre. In *Proceedings of IEEE*, no. 9 in 79 (September 1991), 1305–1320.
16. Hussey, A., and Carrington, D. *Specifying a Web Browser Interface Using Object-Z*. Springer, 1998, 157–174.
17. Le Charlier, B. Abstract interpretation and application to interactive system verification. In *Proc. DSV-IS'96*, Springer (1996), 46–72.
18. Myers, B. Separating application code from toolkits: Eliminating the spaghetti of callbacks. In *Proceedings of the ACM UIST*, Addison-Wesley (1991), 211–220.
19. Myers, B. A., and Rosson, M. B. Survey on user interface programming. In *Proceedings of ACM CHI'92*, ACM (1992), 195–202.
20. Palanque, P., Barboni, E., Martinie, C., Navarre, D., and Winckler, M. A model-based approach for supporting engineering usability evaluation of interaction techniques. In *Proc. EICS 2011*, ACM (2011), 21–30.
21. Thimbleby, H., and Gow, J. Applying graph theory to interaction design. In *Proc. EICS2007*, J. Gulliksen, Ed., vol. 4940 of *LNCS*, Springer Verlag (2008), 501–518.