Verification of properties of interactive components from their executable code

Stéphane Chatty, Mathieu Magnaudet, Daniel Prun

Verification of properties of interactive components from their executable code

Stéphane Chatty **Mathieu Magnaudet** **Daniel Prun**
Université de Toulouse - ENAC
7 av. Edouard Belin, 31055 Toulouse, France
firstname.lastname@enac.fr

ABSTRACT

In this paper we describe how an executable model of interactive software can be exploited to allow programmers or specifiers to express properties that will be automatically checked on the components they create or reuse. The djnn framework relies on a theoretical model of interactive software in which applications are described in their totality as hierarchies of interactive components, with no additional code. This includes high level components, but also the graphics, behaviors, computations and data manipulations that constitute them. Because of this, the structure of the application tree provides significant insights in the nature and behavior of components. Pattern recognition systems can then be used to express and check simple properties, such as the external signature of a component, its internal flows of control, or even the continued visibility of a component on a display. This provides programmers with solutions for checking their components, ensuring non-regression, or working in a contract-oriented fashion with other UI development stakeholders.

Author Keywords

interactive component; verification; properties; static analysis; tree; pattern matching; specification

ACM Classification Keywords

H.5.2 Information Interfaces and presentation: User Interfaces; D.2.4 Software engineering: Software/Program Verification

INTRODUCTION

Using executable representations of programs to verify software properties by static analysis is a well established technique. Static analysis requires no additional modeling work from developers, and can be performed automatically on the source code or on intermediate representations produced during its compilation. With languages such as Java and C it has become a standard step in development processes: programmers rely on the compiler to detect unused arguments, incompatible types, and unusual control flows. Static analysis is also what gives domain specific languages, such as SCADE for command and control systems and VHDL for hardware circuits, a key role in the development of critical systems for fields such as transportation, health and defense.

Static analysis has even been used with languages like Eiffel to introduce software engineering methods based on contracts. Instead of checking generic properties only, the compiler checks properties defined by development teams for individual components. This helps them to organize developments and ensure non-regression.

However, there are limitations to applying these methods to human-centered software such as surveillance systems and vehicle cockpits. Interactive software is software that interacts with its environment, which may include users, pervasive sensors and control systems. This brings additional desired properties, related to how the software will interact. For instance, when working on a graphical button one may want to guarantee that the button will always have three visually different states, and not only that its code will not crash. But various categories of interactive software, including modern user interfaces, have part of their code written with traditional programming languages. This makes the verification of interaction-oriented properties difficult because the native code may have an influence on these properties but it is not accessible to analysis at the desired level of abstraction. In practice, development teams must generally do with traditional computation-oriented verifications.

In this article we present verifications of interaction-oriented properties that are made possible by using an executable language dedicated to interactive software. After reviewing the state of the art we present the djnn programming framework that executes descriptions of interactive components, and show how these descriptions are organized like an abstract syntax tree. We then show how properties can be statically derived from this tree with pattern matching techniques such as XPath. We conclude on the perspectives in terms of methods and tools for developing interactive systems.

STATE OF THE ART

Properties of interactive systems

As stated in [1], an interactive property is a feature of an interactive system that is the subject of analysis and evaluation. During the system definition phase, interactive properties are used to express user needs without any consideration about how they will be fulfilled. They allow external stakeholders

of the system (for example the final user) to capture all features expected from the future system. Properties are also used as drivers for the design: they define the limits in the design space in which the solution must fit. Finally, they are used as a reference for the validation phase during which the final system is checked against the user needs. Such properties are often expressed at task level: *"when activated, the alarm must always be visible by the user"*; *"the user must always have the possibility to abort a task"*.

Another way to use properties has been pointed out in [8], properties are defined during the iterative design of interactive software and used to support and justify design choices. Directly aimed at designers' concerns, properties address technical features resulting from design elements (input devices, displays, etc) and design choices (interaction style). Examples of such properties are *"all mouse press on one graphical component have an effect on another component"* and *"the resolution of the printer is 9600 dpi"*.

In parallel a lot of effort has been spent on classifying interaction-oriented properties. Works initially done in [23], [9] and in software standards have defined several criteria to assess whether a final product meets its expected goals. Criteria focused on usability (learnability, flexibility, robustness) have been added later for graphical user interfaces [1], then multimodal user interfaces [25]. In accordance with [8], these works showed how interaction can be correlated with relationships between design level properties (related to devices, interaction, language) and system level (tasks).

Verification of interactive system properties

During the last decades, several methods for the verification of interactive system properties have been proposed. The empirical method relies on the execution of the final system (or a prototype) with end-users accomplishing selected tasks, in a dedicated environment. During the execution, measurements and observations are performed. Collected data are then used to assess whether the system fulfills the expected properties or not. This approach is widely used for interactive system evaluation and for many properties it is the only available method. The main drawback is its lack of exhaustivity, because properties cannot be checked against all possible executions of the system. This is compounded by the cost of empirical tests, which limits the number of tests that can be run.

Another approach consists of using model-based methods: during the design, a model of the future system is built and properties are verified 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. Several kinds of models can be used according to the nature of the system and the properties to be checked. Formal models are models whose syntax and semantics are defined in an unambiguous way. This allows to define proven model transformations and analysis algorithms that can be used for verifying properties. For example, dedicated algorithms on Petri net models can identify loops within the control flow or prove ressource boundedness; refinement proofs on B models can show that an invariant is preserved when adding more details to a model.

Model checking methods are a class of model-based methods. They are based on the evaluation of logical properties on the state-transition structure obtained from the simulation of the model. For instance a combination of the LOTOS language and the task model Concur Task Tree has been proposed to verify dynamic properties of the user interface [27]. Interactive Cooperative Objects follow a similar approach, using Petri nets to model part of the behavior of interactive components [26]. Safety properties of synchronous languages Lustre and Esterel are also checked with model checking [17, 4].

Alternatively, proof-based methods are a class of model-based methods where the model is described by a set of variables, operations, events, temporal properties and invariants. The operations must preserve these invariants and a set of other properties (preconditions and/or post conditions). To ensure the correctness of these specifications a set of proofs is generated and shall be proved. In the field of interactive software, VDM [13] and Z [18] have been used for defining atomic structures like interactors. The B language has been used to express the design of a system by the way of successive refinements starting from task model level to code level [2]. Finally, logics and type systems have been also applied. HOL (a higher order logic theorem prover) has been used in the verification of user interface specifications [6].

In the context of interactive systems, models are mainly used to express elements with a high level of abstraction. For example many models describe information related to Abstract User Interfaces (AUI). A lot of details related to the concrete interface, including the description of modalities, are not or little included in the models, which makes hard or even excludes the verification of some properties.

Abstract interpretation of software

Abstract interpretation [12] consists in giving an abstract semantics to a programming or specification language so as to perform operations on the abstract representation: property verification, optimisation, debugging. 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.

Static analysis is the major concrete application of abstract interpretation. It is applied by analyzing the source code of programs and automatically extracting an abstract semantic that can be used to verify properties and perform optimisations. This approach has been used in software compilers for a long time (for example [28] uses it for the optimisation of the gcc compiler, [7], [19] and [21] for automatic software parallelisation). This approach has also been used for the verification of various properties of programs: 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. The FLUCTUAT tool [16] is an abstract interpreta-

tion tool for studying numerical C programs, and in particular the propagation of uncertainties in floating-point computations.

There have been some works on using abstract interpretation for interactive properties. [20] first advocated this approach with the objective of providing a unification canvas for verification techniques. [15] described the verification of interactive Web pages through the static analysis of the user interface and ergonomic rules, encoded in UsiXML. [29] proposed to 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. Despite its success, this work illustrates the constraints that often apply to using abstract interpretation on interactive systems: when the system is built with tools that do not capture the appropriate level of information, the missing information must be introduced by producing a model by hand.

In order to support fully automated verifications, one must ensure that what programmers or designers produce contain all the necessary information to derive an abstract interpretation. Any part of the software built with a traditional language foils this approach, because it would be extremely difficult to reconstruct the impact of this piece of code on the program's interactive behavior automatically. Model-based engineering solves this by having humans create task descriptions and deriving the code from this. An alternative consists in providing programmers with an interaction-oriented language that allows them to describe complete applications at an appropriate level of abstraction.

THE DJNN FRAMEWORK

djnn 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 [11]. 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 "hello world" program can be described with two components. The first prints text when activated, and the second binds the activation of the first to the start of the program. Launching the program is an external event that triggers the start of the program, thus triggering the binding component which itself triggers the text-printing component.

djnn has 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.

Creating a program as a hierarchy of components is similar to constructing an abstract syntax tree, as done by traditional compilers. The tree contains all the information needed to execute the program directly or translate it into executable code for a given platform. It also lends itself to static analysis, because all the components of the tree can be interpreted in a common semantic framework. This is what we propose to exploit in the rest of this paper, after giving more details on djnn and its component system.

Root concepts: processes

The conceptual model of djnn can be compared to that of functional programming languages: it relies on a very reduced set of 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].

Like 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. Process-based theories are general enough to model both interaction-oriented software and computation-oriented software [14]. 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. Whole interactive applications can therefore be described in the same language.

Interactive components

Component 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. This includes hardware components as well as software, programmers being in charge of the latter. 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:

- components that support user interaction, either because they are software proxies for hardware devices (e.g. mouse, display monitor) as in [10] or they represent abstractions created on top of hardware devices (e.g. graphical objects, windows, sounds, speech grammar rules);
- components for data representation, such as numerical, text and boolean values;
- computation-oriented components, such as numerical, geometrical and logical operators;

- components that encapsulate pre-existing code written in another language, such as gesture recognition algorithms;
- components aimed at assembling and connecting other components; this includes composite components for grouping, and control structures for connecting components to ensure that they will interact.

Like in traditional software, control structures are the key to providing programmers with the appropriate power of expression. djnn provides the control structures that have been introduced for interactive software in the last decades, as well as traditional computation-oriented control structures. This includes:

- bindings, which ensure simple reactions to events: when two components are interconnected by a binding, activation of the first triggers activation of the second;
- connectors, which ensure that any modification of their input value are propagated to their output values;
- state machines, whose transitions can be triggered by the activation of components, and whose states and transitions can trigger the activation of other components;
- composite components, which propagate their activation to their sub-components;
- iterators, which activate other components in a given order;
- tests, which activate another component only when they are activated and when a boolean value is true;
- switches, which activate one among several components depending on the value of their state.

Programmers can extend this basic set by assembling available components to produce new control structures dedicated to their own needs, for instance control structures dedicated to software adaptation as suggested in [22].

Component hierarchies

Even though programs can be described in their totality by interconnected components, programmers need additional services to manage their code. To start with, when using text-based languages they need to use names to designate the components they wish to interconnect. These names are not essential to program execution and can disappear during compilation, nevertheless they are essential to writing programs. Similarly, programmers need means to organize their components so as to understand how their programs work, organize their tasks, and reuse components between programs.

The composite components of djnn provide support for both organizing components and naming them. Because they contain other components, they allow to create component hierarchies. For instance, a text is part of a button, itself part of a dialogue box, and so on. This defines a natural scheme for naming components: if each component has a name relative to its parent, all components become addressable with names such as `mydialogue/ok/label`. The following pseudocode shows the hierarchy of djnn components for a simple wall clock whose needle turns by 6 degrees every 1000 ms.

```
component wallclock {
  component control {
     clock cl (1000);
     increment incr;
     multiplier mult (0, 6);
     binding (cl, incr);
     connector (incr/state, mult/left);
   }
   component graphics {
      circle (250, 250, 100);
      rotation r (250, 250, 0);
      line needle (250, 250, 250, 100);
   }
   connector (control/mult/result, graphics/r/angle);
}
```

In this example, the `clock`, `increment`, `multiplier`, `binding`, `connector`, `circle`, `rotation` and `needle` are basic components provided by djnn. The `control` component represents the underlying mechanics in the wall clock. Every 1000 milliseconds, the clock named `cl` is activated. Because this clock is bound to the increment named `incr` by an anonymous binding, the state of the increment increases by one each time. Since we want the needle to turn by 6 degrees, the state of the increment is fed through an anonymous connector to the left input of the multiplier named `mult`, and the right input of the multiplier is set to 6. Consequently, the result of the multiplier increases by steps of 6 every 1000 milliseconds. The `graphics` component is a small graphical scene made of a circle, a rotation and a line. Because of its position in the order of the scene, the line is drawn on top of the circle and is rotated by the amount defined in the rotation component. Consequently, the effect of the last anonymous connector is to ensure that the needle is rotated by 6 degrees every second.

Semantics of the hierarchy

The names used in the connectors of the above example, e.g. `control/mult/result`, are representative of how the hierarchy of components is used. This hierarchical structure allows programmers to think of their applications as trees of components, to address components as tree branches, and to reuse whole branches as components in other applications. It also provides the basis for drawing a parallel between the djnn component hierarchy and the abstract syntax trees of traditional languages. In both cases, the execution of the program is defined by a graph of interconnected elements. This graph contains elements that can point at any other element: for instance a function can call itself recursively and a button can trigger the closing of its parent dialogue box, thus creating a loop in the graph. But in both cases, a tree-like subset of this graph captures how the program was organized by programmers. This dual structure is exploited by compilers and by static analysis tools for computation-oriented software. Similarly, we can exploit it to verify interaction properties of djnn components.

With djnn the significance of the tree structure is even higher than usual because, like with XML-based formats such as SVG and HTML and unlike with traditional programming languages, the structure of djnn programs is the same as the structure of their data. This is because interactive applications, and particularly graphical applications, generally have a static structure in which data and behavior are intermixed. For instance, a dialog box is made of a basis of graphics, to

which behavior was added. This combination of data and behavior is so meaningful to programmers and designers that it guided the design of user interface programming tools, including djnn and its component model. At the opposite, computation-oriented programs have a syntactic structure that is guided by software reuse and not by any data structure.

This central role of the tree means that important properties can be evaluated by analyzing it, without need to explore the full graph. In particular, since the introduction of graphical "scene graphs" decades ago the tree structure of interactive applications has been used to express execution control. The position of a graphical object in the tree does not only tell to which component it belongs, it also 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. The shapes added to the right of these style components are then affected by them (see Figure 1). This also applies to geometrical transformations (rotation, scale, translation).

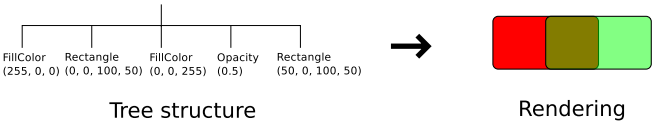

Figure 1. The order in the tree determines the graphical result

The djnn hierarchy of components is an augmented scene graph that contains components as varied as control structures, sounds, files and numerical operators, and sometimes even does not contain graphical objects at all. For consistency, the original context-oriented semantics of the order of components in the tree was maintained and used for other modalities when this is meaningful. This can be easily exploited toward the verification of properties. For instance, as detailed later, it makes it possible to answer questions such as *"will this component be overlaid by another one?"*, *"is the color difference between the background and the foreground components large enough?"*, *"is the display of this message triggered by the right sensor?"* or *"is there an event source connected to each transition of the finite state machine?"*.

Building and exploring the component hierarchy

djnn can be used as a traditional toolkit in which all components are created through a C, C++, Perl, Python or Java API. For instance, the following Java code creates the wall clock described earlier.

```
wallclock = new Component("wallclock");

control = new Component(wallclock, "control");
cl = new Clock(control, "cl", 1000);
incr = new Incr(control, "incr", 1);
new Multiplier(control, "mult", 0, 6);
new Connector(control, 0, incr, "state", mult, "left");
new Binding(control, 0, cl, 0, inc, 0);

graphics = new Component(wallclock, "graphics");
new Circle (graphics, "bkg", 250, 250, 100);
new Rotation(graphics, "r", 0, 250, 250);
new Line(graphics, "seconds", 250, 250, 250, 150);

new Connector(wallclock, 0, control, "mult/result",
                            graphics, "r/angle");
```

But, as illustrated in Figure 2, djnn can also be used as an interpreter for components that are produced with external tools, and all or parts of djnn programs can be loaded from XML files and executed by the djnn interpreter. For instance, the above wallclock can be executed from the following file:

```
<component name="wallclock">
  <component name="control">
    <clock name="cl" period="1000"/>
    <incr  name="incr"/>
    <multiplier  name="mult" left="0" right="6"/>
    <connector in="incr/state" out="mult/left"/>
    <binding source="cl" action="incr"/>
  </component>
  <component name="graphics">
    <circle name="bkg"  cx="250" cy="250" r="100"/>
    <rotation name="r" cx="250" cy="250" a="0"/>
    <line name="sec" x1="250" y1="250" x2="250" y2="150"/>
  </component>
  <connector in="control/mult/result" out="graphics/r/a"/>
</component>
```

The djnn interpreter can be compared to a Java virtual machine, and the djnn XML format to Java byte code. In both cases, executable programs are loaded in memory and run by an interpreter. In this context, the traditional toolkit API of djnn can be considered as an alternative byte code format: components are either stored in XML or in compiled code.

This architecture that separates an interpreter and an executable set of components provides two solutions for performing static analysis of the components. The hierarchy of components can be analyzed in memory, whatever method was used for creating it. This can be done by adding analysis tools to the djnn interpreter. Alternatively, analyses can be performed on XML files using dedicated analytical tools.

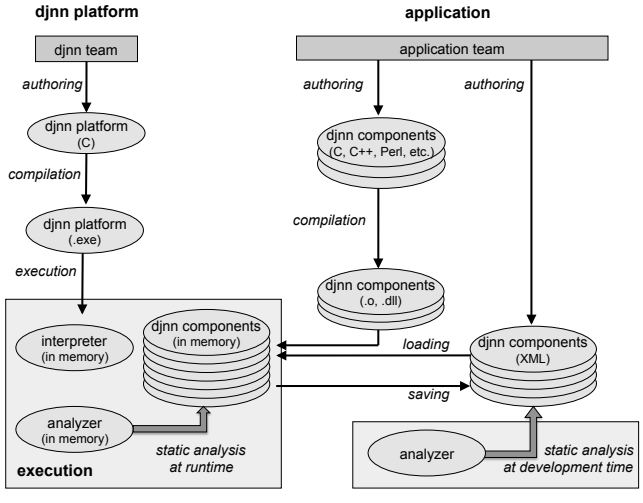

Figure 2. How the djnn platform loads, executes and verifies the properties of components

ANALYZING DJNN COMPONENTS

As stated previously, the hierarchy of components in djnn carries a significant part of the semantics of a djnn program. Consequently, a number of properties can be checked by analysing the tree of components. In this section, we describe several kinds of such analyses. We start with some properties that can be verified by pattern matching in the tree only. For practical reasons, we use XPath queries, although other methods may prove useful in the future. We then move to properties that need additional more sophisticated graph analysis.

Pattern matching with XPath

XPath is a language specified by the WWW Consortium [30], originally dedicated to the exploration of XML trees. It defines a rich grammar to build expressions that can be used to select nodes in a hierarchical document. The syntax of an XPath expression can be quite complex. It offers various constructs to specify a search path (relative, absolute), to express relationships, called axes, between the nodes (parent, children, sibling, descendant, etc.) and to check properties of the elements or their attributes through logical and arithmetic expressions. Processing an XPath query results in a list of items that all contain elements that match the expression. For example the expression `expr=(/widget/descendant::*/button)` retrieves all the "button" elements that descend from the "widget" element.

Although XPath was designed for XML, it can be used with any hierarchical structure that is similar enough to the ontology of an XML document: element, element value, attribute, attribute value, etc. This is the case of the hierarchies of components in djnn, both in their XML form and when loaded in the djnn interpreter. When executing a djnn program, two phases can be distinguished: the building of the hierarchy of components in memory, then its execution. We focus on the first phase, in which djnn can perform static analysis automatically before running the program. This provides a convenient time, if not always optimal, to provide developers with feedback on the robustness of their code.

Verifying component signature

Most modern developments involve teams of developers that build or evolve components in parallel then combine them. In the case of graphical user interfaces, this concurrent engineering can even involve different professions, with graphical designers producing the visual components and programmers producing the rest. The efficiency of this development process would be improved if all developers could verify before integration that their components follow the contract that was originally decided upon. Here, a number of such contracts can be defined as the existence of a specific pattern of descendants in a component.

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)` do not return a null result.

Similarly, before adding a drag and drop behavior to a component, we must check the existence of the `press/x` and `press/y` patterns, that is the event and properties that will trigger the behavior, as well as the existence of and `x` and `y` children, that is the properties that will be changed by the behavior. The corresponding XPath requests are, respectively, `expr=(/component/press/x)`, `expr=(/component/press/y)`, `expr=(/component/x)` and `expr=(/component/y)`.

Patterns can also be used in a more flexible way. Suppose that one wants to build a clock with three needles for hours, minutes and seconds. Coding the clock behavior involves triggering a periodical change of the angle of the needles around their respective rotation center. This code can only work if the graphics have three needles. A good way to verify this condition is to check it through an XPath request: the graphical component must contain components called `hours`, `minutes` and `seconds`. XPath allows to find a component without assuming any specific place for it in the tree. For instance, the expression `expr=(//hours)` will find, if it exists, the `hours` component whatever its position in the graphical component. Note that the condition is verified for many different graphical design (e.g. figure 3). Thus, this verification focuses on what is needed exactly and does not impose a strong restriction over the power of expression.

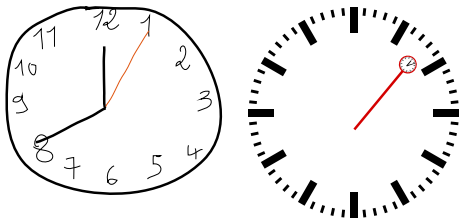

Figure 3. Two graphical designs for a clock verifying the same condition

Verifying execution context

The specification of an interactive system sometimes includes the expression of constraints on the hardware and software environment in which a given application will run. For instance, some applications require a minimum screen size to ensure readability and others require a specific input device to support a specific interaction style. Some of these constraints can be checked through XPath requests over the extended djnn tree, that includes the context.

The djnn framework gives programmers the possibility to explore an extended component hierarchy that represents the context in which their program runs. Their own components, when created, become part of this extended tree. Upon request, the extended tree can contain the available physical displays, the input devices, the batteries, etc. Once initialized, the extended tree can be explored like the application tree and its components can be used in control structures in the application tree. Consequently, verifying that a large enough physical display is available amounts to verifying the existence of a specific component in the display tree through an XPath request such as: `(/displays/display[@width > 800])`. Figure 4 shows how the result of the query can be used to decide

whether or not to launch a program depending on the current hardware configuration.

```
root = new Component ()
d1 = Element.findElementByXpath (displays, "display(@height>800)")
if (d1 == null)
   abort()
root.addChild ("Display1", d1)
new Frame (root, "f1, ...)
d2 = Element.findElementByXPath (displays, "display(@width>800)")
if (d2 == null)
   abort()
root.addChild ("Display2", d2)
new Frame (root, "f2", ...)
```

CODE

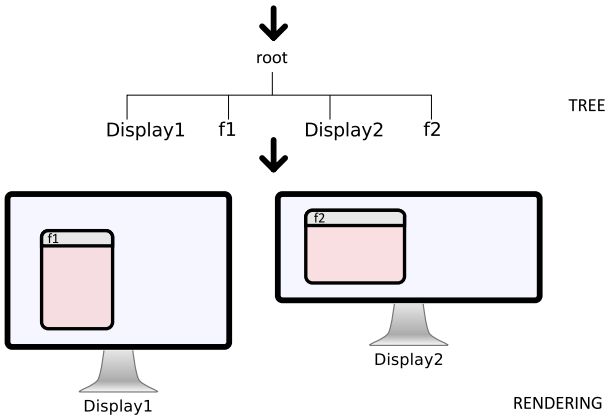

Figure 4. Checking the presence of an appropriate physical display

Verifying component visibility

As mentioned earlier, the order of graphical components in the tree determines what will effectively be displayed. Based on this, it becomes possible to verify some properties such as the visibility of a graphical component. The properties to be verified are, at least, that the component is rightmost in the tree and that there is no opacity components with a zero value on its left. Moreover we have to check that, if an opacity component is present in the context, there is no event source that can modify it. The following example illustrates such a situation:

```
<component uri="djnn://root">
  <clock name="cl" period="100"/>
  <incr name="incr" />
  <formula name="f" formula="a = (1-i/100)"/>
  <connector in="djnn://root/incr/state"
             out="djnn://root/f/i"/>
  <binding name="b" source="djnn://root/cl"
                    action="djnn://root/incr"/>
  <frame name="frame" title="myFrame" x="50" y="50"
                      width="250.00" height="250.00"/>
  <fill-color name="fc" r="30" g="10" b="200"/>
  <fill-opacity name="fo" a="0.81"/>
  <connector in="djnn://root/f/_child3/result"
             out="djnn://root/fo/a"/>
  <rectangle name="myR" x="10" y="10" width="150"
                        height="100" rx="0.5" ry="0.5"/>
</component>
```

This piece of XML is a complete representation of a simple program. The program displays a blue rectangle, and a clock periodically triggers an increment that is used through a calculation to modify a fill-opacity component (Figure 5). We want to check if the rectangle is always visible.

It is quite easy to see, on this example, that the rectangle is the last graphical shape of the tree. We can also automate this verification, by combining XPath queries in a simple algorithm. Firstly, we have to check the very existence of the graphical object of interest. If the object's name is `top` then we can check that the expression `exp = (/descendant::*[@name='top'])` does not return a null result. Then we have to verify that no graphical shape can mask it. The method consists in retrieving the list of all the graphical shapes situated in the tree after the closing tag of the `top` component. This list can be retrieved with a series of XPath expressions of the form:

```
/descendant::*[@name='top']/following::*/rectangle
```

where the last name is replaced by the different kinds of graphical shapes (rectangle, circle, path, etc.). For each expression, it must be verified that the returned list is empty.

The second step consists in checking that there is no opacity component at the left of the component. Similarly, with the expression:

```
/descendant::*[@name='top']/preceding::*/fill-opacity
```

it is possible to verify if there is an opacity component that can change the visibility of the component `top`. Of course, the opacity value can be high enough to guarantee the visibility of the following components. But we have to detect if the value can be changed during execution. Opacity components have a property named `a` whose value fixes the current opacity. Thus, if the name of the opacity component is `fo`, then we have to check if there is a data flow component, a connector, whose output value ends with `fo/a`. Once again we can build an XPath expression to retrieve all the connectors of the tree:

```
/descendant::*/connector
```

Then, with a simple algorithm, it is possible for each node to retrieve the value of its output attribute and to verify whether or not it ends with the specified suffix. If this is the case, this means that the property can change during execution, and that the visibility of the component cannot be ensured.

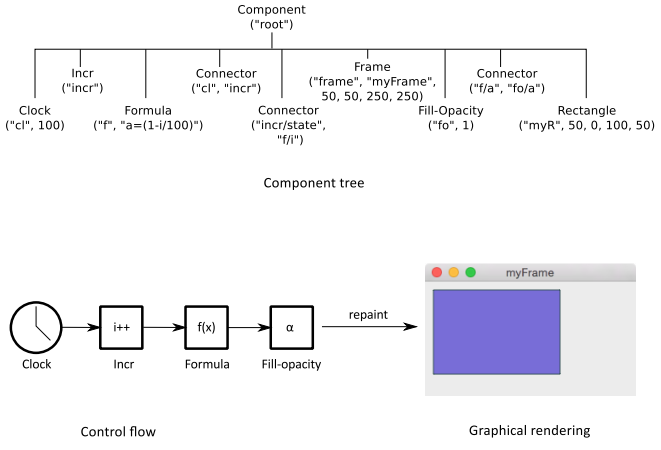

Figure 5. Various representations of an animated program

Verifying the control flow

The combination of XPath queries to select elements and simple algorithms over their results allows to address more complex cases. The example shown in Figure 6 is a classic cockpit component, the primary flight display, here represented with an alarm message on the top. This message alert must be triggered by a proximity sensor that has two children `true` and `false`. The message must be displayed when the `true` child of the sensor is active and must disappear when the `false` child is active.

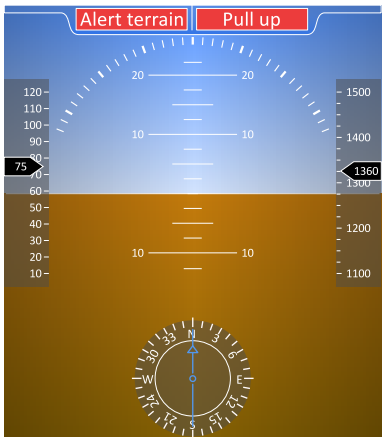

Figure 6. A primary flight display for aircraft cockpits

The usual way offered by djnn to program this behavior is to combine a switch component and a finite state machine (FSM). Any number of children can be added to a switch but only one is active at a given time; which child is active is defined by the state of the switch. When the state of a FSM is connected to the state of the switch, the FSM controls which child is active. In our case, adding the alarm message to one branch of a switch ensures that it is visible when the FSM decides that this branch is active. This ensures that the visibility of the message is controlled by events received by the FSM.

Consequently, to verify the behavior of the alarm we have to check at least that:

1. the alarm message component is a child of a switch;
2. the switch is connected to a FSM;
3. the FSM is well formed;
4. there is a bijection between the branches of the switch and the states of the FSM;
5. the transition in the FSM that goes to the alarm state is triggered by the right event source.

Regarding point 1, the parent of an element can be easily retrieved through the following XPath expression:

```
expr = //descendant::*/component[@name='terrain_alert']/parent::node()
```

Once found, we can check that it is actually a switch through the attribute `name` of the node. For point 2, we have to verify the existence of a connector that connects a FSM to the switch found previously. This can be done by chaining XPath requests. The first finds all the elements in the tree whose node type is `connector`. Then we can look for a connector whose `in` attribute corresponds to the state of a `fsm` node and `out` to the state of the switch.

If a connected FSM is found, we can make classic verifications on it (point 3) such as "is there a transition allowing to reach each state?". This means here:

```
S = XPath.evaluate ("/fsm/children::*/state")
T = XPath.evaluate ("/fsm/children::*/transition")
for each state s in S verify that:
  exists t in T such that output (t) = s
```

We also have to check that for each branch of the switch, there is a corresponding state in the connected FSM (point 4). For this, we must verify that for each branch of the switch, it exists a state in the FSM with the same name. Finally, to verify point 5 we have to follow the control flow that goes from the transition whose output is the alert state, to the event source that will ultimately triggers it. This step can be seen as a partial traversal of the control flow graph. It consists in collecting the control nodes of the tree and in exploring the input and output nodes that they connect.

We implemented the final verification as a Java test that runs the XPath queries on the XML version of the program, then executes the above steps. In our program, there is a connector `c1` between the switch controlling the message and a FSM. The transitions of the FSM are connected to the `terrain_alert` and `end_terrain_alert` components, themselves connected to the `true` and `false` children of the proximity sensor through two bindings components `b1` and `b2`. The result is the following:

```
[connector:c1] connects djnn://root/fsm/state
                    to djnn://root/sw_alert/state
[fsm:fsm] well formed
[fsm:fsm] and [switch:sw_alert] rightly connected

Transition from alert to no_alert
          triggered by event djnn://root/end_terrain_alert
[binding:b2] connects djnn://root/proximity_sensor/false
                  to djnn://root/end_terrain_alert

Transition from no_alert to alert
          triggered by event djnn://root/terrain_alert
[binding:b1] connects djnn://root/proximity_sensor/true
                  to djnn://root/terrain_alert
```

Thus we can verify that the alert message will indeed be displayed only when the `true` child of the proximity sensor is activated.

PRACTICAL APPLICATIONS AND PERSPECTIVES

djnn is available at **`http://djnn.net`**. It was designed to support the development of modern interactive systems, including post-WIMP and multimodal interaction, ubiquitous computing and internet of things applications, and to explore new engineering processes for interactive software. Since practical application to real life developments is essential to the validation of the underlying research, djnn is maintained and proposed to user interface developers on several operating systems. It comes with a complete graphical module, and with partial support for modern input devices, multiple displays and file management. Other modules are under development for supporting various interaction modalities, all

based on the theoretical principles explained earlier and thus all accessible to static analysis.

djnn has been used to develop various real-size applications including a touch-based ground control station for squads of drones, a pen-based map annotation system for crews of search and rescue missions, a multimodal cockpit prototype for civil aircraft, and an interactive show room. Some are illustrated in the djnn web site. All these applications have been designed and implemented using the same process: user-centered design methods yielding low fidelity prototypes (mostly paper prototypes, sometimes video prototypes), parallel production of graphical components and behavior components, and final integration. The graphical components are produced with Inkscape or Adobe Illustrator and loaded from SVG files when the application is initialized, while the rest of the component hierarchy is created through the djnn APIs. The most popular APIs are the Java API and a proprietary ObjectiveC API created by a user interface design company. The main application of the XML formats so far has been to transform or verify programs that had been previously created by hand, not to execute models produced through controlled processes. This underlines the utility of performing property verification on executable programs without making assumptions on how these programs have been produced.

The verification of properties on djnn components is still experimental work and has not yet been integrated in the public releases. As explained previously, some properties are currently checked on the representation of the tree in memory when programs are launched and others on XML files using external tools. Each of these two approaches has proved useful. The external approach is useful for exploring properties and algorithms before considering an implementation in djnn. But it also helps to imagine solutions in which verifications are performed offline when modifications of programs are committed to version management systems or continuous integration systems. As for the internal approach, it also opens the door to verifications that would be performed at run time and not only when the program is initialized. For instance, verifications on the context such as checking what kind of display is available can be executed when the list of available displays changes, thus becoming part of the interactive program itself as proposed in [22].

In both cases, what the examples in this article illustrate is that interesting properties can be defined at the level of each component by programmers. There probably are general properties than must be checked for all programs, like what compilers propose for traditional programming languages. But, in the same vein as the verification of invariants in the programming-by-contract paradigm [24], there are many properties that are meaningful for a given component only, or for a given development team. This justifies supporting the management of such properties in languages and tools for interactive software. For instance, the XML format for djnn components can be enriched with new sections for each component, as in the example below:

```
<component name="graphics">
  <xpath-required query="//sec" />
  <circle name="bkg"  cx="250" cy="250" r="100"/>
  <rotation name="r" cx="250" cy="250" a="0"/>
  <line name="sec" x1="250" y1="250" x2="250" y2="150"/>
</component>
```

These conditions, evaluated at verification time rather than run time, can be defined by designers or development teams before starting development, as a materialisation of a contract. Or they can be added by programmers during development, because they notice that they had to guarantee them even though they were not originally specified; doing so ensures non-regression for the future.

CONCLUSION

In this paper, we have developed an approach for the verification of interactive system properties based on static analysis and on the use of a language that describes interactive software as a hierarchy of interactive components. This approach transfers to interactive systems a method that is well known and used in other software domains. It gives the djnn framework the ability to support the formulation of component properties by members of development teams. Building abstract interpretations of djnn programs allows to check various properties automatically and directly on the code, whether they are related to design level or to system level.

So far we have studied abstractions based on the component hierarchy, and more marginally on the control flow graph. Even if promising results have been obtained, a lot of further research is needed to explore the potential of this approach. Several research directions are currently under consideration. The range of properties expressible with XPaths must be explored as well as other methods for pattern matching in trees, including easier languages for expressing queries. It will also be useful to explore how well known graph properties and related algorithms can contribute to establishing useful properties, notably design structure matrix, graph traversal algorithms and graph covering problem. Another possible research area is how other abstractions built from the djnn model can be used for verification, for instance the data flow graph and the slicing approach.

Finally, we must explore what kind of design and development processes can be introduced for managing interactive software using these new possibilities. Contract-based development is an option, as well as incremental specification in parallel with prototype development. The extension of the proposed method to the verification of the compatibility of software with users' activities and tasks can also be explored.

As a final note, it should not be forgotten that it will be some time before the effective perception of an alarm can be proved, and not only its effective display. Empirical evaluation remains an important validation step for many systems.

ACKNOWLEDGMENTS

This research was partly supported by the French DGAC through project FUMSECK and the EU ARTEMIS Joint Undertaking through project HoliDes (SP-8, GA No 332933). Any contents herein reflect only the authors' views. Neither DGAC not the ARTEMIS JU are liable for any use that may be made of the information contained herein.

REFERENCES

1. Abowd, G. D., Coutaz, J., and Nigay, L. Structuring the space of interactive system properties. In *Proc. IFIP EHCI'92*, North-Holland (1992), 113–129.
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. Bumbulis, P., Alencar, P., Cowan, D., and Lucena, C. Validating properties of component-based graphical user interfaces. In *Proc. DSV-IS96*. Springer, 1996, 347–365.
7. Burke, M., and Cytron, R. Interprocedural dependence analysis and parallelization. In *Proc. SIGPLAN'86*, ACM (1986), 162–175.
8. Campos, J., and Harrison, M. The role of verification in interactive systems design. In *Proc. DSV-IS98*, Springer (1998), 155–170.
9. Cavano, J. P., and McCall, J. A. A framework for the measurement of software quality. In *Software quality assurance workshop on functional and performance issues* (1978), 133–139.
10. Chatty, S., Lemort, A., and Valès S. Multiple input support in a model-based interaction framework. In *Proc. of the 2nd IEEE workshop horizontal interactive human-computer systems (Tabletop'07)*, IEEE Computer Society (2007), 179–186.
11. 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.
12. 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.
13. Duke, D., and Harrison, M. Abstract interaction objects. *Comput. Graph. Forum 12*, 3 (1993), 25–36.
14. Goldin, D., Smolka, S., and Wegner, P., Eds. *Interactive computation - the new paradigm*. Springer Verlag, 2006.
15. 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.
16. Goubault, E., Martel, M., and Putot, S. Asserting the precision of floating-point computations: A simple abstract interpreter. In *ESOP 2002*, vol. 2305 of *LNCS*, Springer (2002), 209–212.
17. Halbwachs, N. and Caspi, P. and Raymond, P. and Pilaud, D. The synchronous dataflow programming language Lustre. *Proc. of the IEEE 79*, 9 (1991), 1305–1320.
18. Hussey, A., and Carrington, D. *Specifying a Web Browser Interface Using Object-Z*. Springer, 1998, 157–174.
19. Kennedy, K., and Allen, J. R. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2002.
20. Le Charlier, B. Abstract interpretation and application to interactive system verification. In *Proc. DSV-IS'96*, Springer (1996), 46–72.
21. Lim, A. W., and Lam, M. S. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proc. POPL'97*, ACM (1997), 201–214.
22. Magnaudet, M., and Chatty, S. What should adaptivity mean to interactive software programmers? In *Proc. ACM EICS'14*, ACM (2014), 13–22.
23. McCall, J. *Factors in Software Quality: Preliminary Handbook on Software Quality for an Acquisiton Manager*, vol. 1-3. General Electric, November 1977.
24. Meyer, B. *Object-Oriented Software Construction*. Prentice Hall, 1988.
25. Nigay, L., and Coutaz, J. *Multifeature Systems: The CARE Properties and Their Impact on Software Design*. 1997.
26. Palanque, P., Barboni, E., Martinié, 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.
27. Paternò, F. Formal reasoning about dialogue properties with automatic support. *Interacting with Computers 9*, 2 (1997), 173–196.
28. Pop, S. *The SSA Representation Framework: Semantics, Analyses and GCC Implementation*. PhD thesis, Ecole des Mines de Paris, 2006.
29. 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.
30. W3C. XML path language (XPath) 2.0 (second edition). `http://www.w3.org/TR/xpath20/`.