What should adaptivity mean to interactive software programmers?

Mathieu Magnaudet, Stéphane Chatty

▶ To cite this version:

Mathieu Magnaudet, Stéphane Chatty. What should adaptivity mean to interactive software programmers?. EICS 2014, ACM SIGCHI symposium on Engineering interactive computing systems, Jun 2014, Rome, Italy. pp 13-22, 10.1145/2607023.2607028 . hal-01011095

HAL Id: hal-01011095
https://enac.hal.science/hal-01011095

Submitted on 23 Jun 2014

What Should Adaptivity Mean to Interactive Software Programmers?

Mathieu Magnaudet
Université de Toulouse - ENAC
7 avenue Edouard Belin
31055 Toulouse, France
mathieu.magnaudet@enac.fr

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

ABSTRACT
Works about adaptability and adaptivity in interactive systems cover very different issues (user adaptation, context-aware systems, ambient intelligence, ubiquitous computing), not always with the explicit goal of supporting programmers. Based on examples that highlight how weakly discriminative the present terminology is, we propose to separate two concerns: adaptivity as a purely analytical concept, relative to a given viewpoint on the software rather than to its very structure, and its programming as a non specific case of reactive behavior. We describe how simple adaptive behaviors can be programmed with simple interactive behavior patterns, and how more complex patterns can be introduced for intelligent adaptation. Finally we describe an application where, relying on the principles exposed in this paper, interaction and adaptation are combined in a simple and innovative manner.

Author Keywords
Adaptive software, plasticity, responsive design, context-sensitive applications, software architecture, programming, theory of interactive systems.

ACM Classification Keywords
H.5.2. Information Interfaces and Presentation: User Interfaces: D.2.11.Software engineering: Software Architectures

INTRODUCTION
Software adaptivity is a long time concern in the research domain of interactive computing systems that can be traced back at least to 1975 and John H. Holland's seminal work [22]. In the early 80s, the adaptivity of user interfaces, broadly understood as the self-modification of a system under context variations, arose as a significant issue in the improvement of usability [23]. However subsequent works in this domain forked in various directions depending on the context considered: while early works focused on user adaptation and user modeling [29], the spreading of new execution platforms, such as PDAs, tablet PCs and smartphones, shifted the focus to interaction devices [36, 16]. In the meantime, the generalization of small sensors in computing systems (light, temperature, humidity, etc.) raised the more general issue of adapting software applications to their physical environment [21]. At the same time, progress in distributed architectures brought forward the issue of adaptation to new services or new software components in the execution context of an application [6].

During this process a number of sub-communities addressing specific issues have emerged: user modeling, sensor modeling, middleware, adaptation policy, model-based adaptation, to cite but a few. Each sub-community has developed their own concepts and vocabulary (e.g. adaptation, plasticity, context-aware application). From a theoretical point of view this situation is quite unsatisfactory: if we agree that the science of human-computer interaction is also a theoretical endeavor and cannot be reduced to a collection of methods or good practices, we have to work toward the clarification of its basic concepts and progress toward their unification.

But this is not simply a matter of theoretical aim. Heterogenous concepts yield heterogenous software tools, introducing unnecessary complexity for programmers. Developing an adaptive software that includes several of the dimensions cited above (natural environment, input devices, user's cognitive abilities for example) can be a daunting task. This might explain why the computing industry has introduced "responsive design", a less ambitious but more practical concept for adapting interactive applications to their execution environment. If we want support for adaptivity to find its way into operating systems, like it happened for touch interaction and gesture recognition, eliciting concepts that are simple enough to be embedded in programming tools is a key step.

This article is a contribution toward this goal. We show that the entanglement of programming issues and adaptivity concepts from the state of the art can be untangled into separate concerns: an analysis framework of software adaptivity on the one hand, a set of simple programming concepts on the other hand.

Firstly, we propose a new analysis framework for software adaptation. We show that there is no clear cut division that emerges from previous works between programming adaptation and programming interactive behavior. We propose a definition in which adaptivity appears as quality of interactive behavior, that often has no particular relevance to programming. We then turn to the practical consequences of our definition. We obtain confirmation on standard adaptation situations that no dedicated primitives are required to program adaptive processes: the simple "event - control - action" schema can be tailored for this purpose. Finally, we further validate these principles by demonstrating them on a full-size concrete application, using an existing reactive programming framework.

PREVIOUS WORK

Adaptation to the user

Adaptive interfaces emerged as a research topic in the 1980s (cf. [29] for a review). Simply stated, the idea was that the software must adapt to users rather than the opposite. Users were mostly considered for their cognitive skills, and the topic was strongly associated to cognitive ergonomics. According to Greenberg and Witten [18] for instance, the condition of interface adaptation is that the software manages a model of its users. They proposed to define adaptation as the automatic transformation of this model during the use of the software. They also introduced a now common distinction between automatic transformations and transformations operated by the user via configuration parameters, or by a designer in a process of reengineering.

In this context, adaptive user interface were considered a consequence of the automatic transformation of a user model. The main difficulty was to find the rules allowing to infer the (cognitive) state of users from their actions. This goal created a clear connection between adaptation and the field of artificial intelligence [27]. In more recent works, the focus has moved toward neuroscience with the introduction of physiological sensors and neuroimaging [31, 34]. With these techniques, the user model has been enriched with new dimensions such as affective state [37] or stress level [20], but the principles of adaptation remain the same.

Adaptation to the context

With the 1990s and the increasing variety of computer form factors (larger display sizes, then PDAs, mobile phones, tablet PCs), and more recently new sensors (acceleration, light, pressure, humidity, etc.), new adaptation concerns appeared. We can distinguish at least three of them: adaptation to the execution platform [36, 28], adaptation to the environment [33], adaptation to the applicative context [30]. These various dimensions of adaptation are often gathered under the general expression "context-aware systems". Many of these works propose model-based solutions to the problem of context variation. Models range from complex ontologies [12, 19] to more partial models of the context or of some parts of the software system [17, 13]. They are usually associated with algorithms, inference rules or policies specifying how to modify an application according to the modeled context dimensions (cf. [3] for a review).

There is no consensus, however, about what exactly must be included in the context of an application: authors each choose their own focus of interest and propose their own characterization of what they take to be the relevant context of an application. This ranges from technicalities to topics that are closer to user-centered adaptation: physical properties, surrounding objects, user emotional state, etc.). By contrast Dey and Abowd [14] propose a general definition:

> *Context:* any information that can be used to characterize the situation of entities (*i.e.*, whether a person, place, or object) that are considered relevant to the interaction between a user and an application, including the user and the application themselves. Context is typically the location, identity, and state of people, groups, and computational and physical objects.

This definition clarifies the common focus of all these works: the interaction between the user and the software. It is pragmatic and flexible, with the drawback that it does not provide an absolute definition, independent of any application.

Adaptation in software architecture

Another area of research is more focused on the pragmatics of implementing software. Historically this was first documented when the domain of interactive software architecture emerged in the 1980s from the need to adapt existing applications to graphical user interfaces. The Dialogue layer in the Seeheim model [32] partly serves this purpose. The Functional Core Adapter in the Arch model has the explicit role of permitting the adaptation of a functional core to various interaction layers, and reciprocally [4].

Note that this is quite remote from the definition proposed by Greenberg and Witten for adaptation: here, adaptation is not even a matter of configuration parameters but of sheer reengineering. The adaptation is not relative to the user or the environment, but to the programming interface of software components. It is only recently that connections have appeared with context adaptation, with the proposal to handle software components and input devices through a unified component model [10].

Meanwhile, this type of adaptation has become a general issue in software engineering. Software reengineering, Web services and ubiquitous computing each have brought their own needs, leading to various solutions. This even includes network security, where adaptive architectures have been proposed to manage the variations in the security level of physical networks [5]. Solutions such as aspect-oriented programming [24] or component-based architectures (Enterprise Java Beans, Corba Component Model) are related to this issue of supporting programmers who manage the adaptation of software components [26].

Lessons learned

From this overview of previous works we propose two lessons. The first is that adaptivity encompasses a huge variety of works, especially when context and context-aware systems are understood in such a wide sense as in Dey's and Abowd's definition. Building a set of core concepts that can be derived for each kind of adaptation could help to offer more integrated support for adaptivity.

The second lesson is that, in contrast with what happened in the software engineering field, research on adaptation in interactive software has been carried out in relative independence from research on programming concepts. Studying how the core concepts of adaptivity relate to those of interactive software programming could also be beneficial.

In the next two sections we study these two lessons in more detail so as to propose an analysis of adaptivity and its relations to interactive software programming.

ADAPTIVITY, A MATTER OF POINT OF VIEW

From the above overview, can we derive a definition of software adaptivity that may be used for providing support to programmers? For this, we need to overcome a few difficulties and come back to the roots of the concept of adaptivity.

Software adaptivity: from a fuzzy concept...

A closer look at the concept of software adaptation as it is presented in the literature reveals some ambiguities. The first resides in the distinction between the transformations that are applied automatically and those that are applied by the user via a configuration menu for example. Only the former are usually considered as adaptations *proprio sensu*, but sometimes the difference can be tenuous. In [34], for example, the authors investigate the possibility to detect the mental load of the user so as to adapt the user interface accordingly. But, interestingly, other works propose to use the exact same technique as a means to add an explicit control device to a system, *i.e.* as a new input modality [38]. Technically and from the information processing point of view, there is no significative difference between these two situations: in both cases the goal is to enable the software to react to a specific brain process detected by a neuroimaging device. The difference lies in the voluntary versus involuntary nature of a specific brain process of the user; the former will be classified as interaction the latter as adaptation. Equally ambiguous situations can be found with eye tracking, RFID or movement detection. In these cases, the difference is more in the eye of the observer of the human-machine interaction than in the software itself. This raises an even wider question, that of the actual difference between adaptivity and plain interactivity: if the same piece of interactive software can be used for both, what use is the distinction to programmers?

Even if we take the more relaxed definition of context adaptation, we are faced with similar difficulties. While this is not often stated in the literature, it is useful to remind that context is a relative concept: it can only be defined with regard to something, a task or an event for example. For adaptive user interfaces, the context is usually that of the current activity of the user and, taken in a wide sense, it encompasses everything that can affect this activity. But, here too, the difference between an activity and the context of this activity seems to lie very much in the eye of the observer (the user, the designer, the scientist). Consider for instance the accelerometers that are now embedded in most smartphones. They typically allow to adapt the graphical interface to the orientation of the phone, but they are also used as an input modality in many applications, especially games. Thus what was considered as a ”context sensor” for some activities is also an input modality for other activities. Once again, not only is it difficult to come up with a clear cut application of the proposed definition, but we also have two processes that are similar from from the programmer's point of view and different from other points of view.

... to a relative concept

The above examples emphasize how much the categorization of a process as an adaptive one depends on the chosen point of view on the human-machine system. The distinction works well enough if we consider users and their activities, or what a given designer takes to be the users' task, or the human-computer interaction itself. From these points of view, adaptivity encompasses all transformations that are not directly caused by deliberate actions of the user. The other transformations, voluntarily triggered by the user, are considered as classic cases of interaction.

On the contrary, from the pure software point of view there is no difference. Programmers are concerned by the interactions of their software with its whole environment, not only with the user. May the action be voluntary or not, the intention explicit or not, the process can always be described as an association between an event and a transformation: detection of a change in the environment (user's actions included), then modification of the state of the software.

Thus, if one is interested by the usability of a system, the concept of adaptation can be useful as an analysis tool for the processes that surround the user's activity. But for those involved in the programming of these processes, this concept does not seem to offer very much.

Alternatively, the distinction is sometimes founded on what is considered to be the function of a software system. In these cases, adaptation characterizes software transformations that maintain its main functionality. This is perfectly illustrated by the works on plastic interfaces [9], where transformations of the user interface are oriented toward the maintenance of the functionality of the software. Here too, adaptation offers an analysis tool to identify what must be transformed and when. However this tells little about how to program such processes.

Also note that other points of view on systems are possible. Consider for instance an air traffic control room with a number of “open positions”, each consisting of a workstation and two operators. The control room, considered as a system, adapts to the level of traffic: when the level increases, controllers trigger de-grouping transformations in which the software helps them to move part of the traffic to newly opened positions. This makes the whole human-computer system adaptable to its context. What is particularly interesting here is that a system engineer would perceive this as an adaptation process, while neither the programmers nor the users or the interaction designers would. Still, the software needs to support it.

The notion of point of view seems crucial to understand adaptive processes in interactive software. This will appear more clearly by analyzing the very concept of adaptivity.

Back to the roots

Adaptivity is a concept inherited from biology. It refers to the ability of a system to self-modify according to the evolutions of its environment in order to maintain or enhance its viability. Adaptivity is not an all or nothing phenomenon but rather a continuum. Systems are more or less adaptive, depending on both the variation range of the environmental properties and the number of these properties. Moreover, not all evolutions are viable; adaptation is a process of transformation under constraints that must maintain the system in its so-called viability kernel [2].

The fact that adaptive processes are oriented toward an end (*i.e.* the maximisation of the viability of a system) is a crucial feature of this phenomenon. Indeed, this is what distinguishes adaptation from a simple mechanical transformation. Thus, the expansion of an iron bar caused by heat is not considered as an adaptive processes. But the thermoregulation mechanisms of the living organisms are considered as such because they are aimed at maintaining properties that are essential to survival. Adaptivity of artificial systems obeys to the same schema with the noticeable difference that viability criteria are not intrinsic but defined from the outside, *i.e.* by the designer, the human factors expert, the user, or any stakeholder involved in the lifecycle of the system.

A new definition

Therefore, we propose to define adaptation as *a function that maps changes in an environmental state affecting the viability of a system to evolutions in the state space of the system (i.e. transformations) in order to maintain its viability*.

Here, environment and transformation are taken as observer-relative concepts. Each observer chooses to delineate a system and its boundaries, the environment is everything that is outside these boundaries, and transformations are all the changes within the boundaries. For programmers, the system consists of the software components they are in charge of, the environment is everything else, and the transformations are the modifications of the software components and their data. These transformations may come out as graphical changes, behavior changes, etc.

With this definition, software adaptivity clearly appears as an external property that is relative to the criteria chosen by each stakeholder to characterize the viability of a system. For instance, from the point of view of the usability expert the viability will be assessed against the usability criteria. Alternatively, the software architect may characterize the viability of a software as its ability to support updating processes, or the addition of new components.

We are now able to explain why the same transformation process may be characterized in the same time as an adaptive one and as a simple interaction: it simply depends on the analysis criteria. The automatic change of the luminosity of a screen according to the variation of the ambient light is a simple interaction in the eye of the programmer. But it is also an adaptive process for the ergonomist who studies how the ambient light affects the visibility of a graphical component (and then, the viability of the software) and how the change of the screen luminosity can correct this issue.

Moreover this definition allows to qualify the roles of various stakeholders in the engineering of adaptive software. Usability experts or systems engineers are interested by the viability of a transformation, that is by the relevance of a software transformation for the maintenance of its function. Designers are interested in the exploration of the transformations space. Programmers are more interested in the means to observe the environmental state and to relate it to a set of transformations. Adaptivity becomes an explicit concern for them only in specific cases. This may be when the adaptation criterion is embedded in the software itself, and they need to implement it. This is typically the case of learning algorithms that have to check whether a transformation is successful. Such software could arguably be qualified as “self-adaptive” software.

Finally, this definition is broad enough to include the cases of software adaptation triggered by software or platform evolutions. For example the adaptation of an x86 application to an ARM architecture and the refactoring of a component to adapt it to a new version of a protocol, match the definition as much as the resizing of a window to adapt to a new display.

PROGRAMMING ADAPTIVE SOFTWARE

The above definition is general enough to encompass the whole range of adaptive phenomena. It also clarifies the distinction between a process of transformation and the finality of this process. From this, we can infer that the programmer is more concerned by the building of the transformation process than by the specification of its finality. But we still need to understand how programmers build the transformation process into their programs: what are their responsibilities, what support do they need? For this, we propose to use an analysis framework specifying the dimensions that structure this space.

Extending an analysis framework

In [36], Thevenin and Coutaz propose a framework that characterizes adaptation along four axes: actor, time, means, target. However, if adaptation is a relational property that depends on a specific point of view on the mapping from environmental changes to system transformations, then the framework must be adapted to reflect this.

We propose to consider that software transformations are themselves particular cases of software actions, along with interactive behaviors such as beeps or color changes. Calling “interaction” or “transformation” the effect of an action results from the same choice of point of view as above. In other words, *all the software transformations considered for adaptation are reactive behaviors and can be analyzed in the unifying framework of reactive processes*. Consequently, we propose to consider that the framework describes reactive processes in general, and to supplement it with an axis representing the possible observers of the system and their adaptation criteria. It is along this axis that adaptive processes can be distinguished from simple interaction.

The other dimensions of Thevenin and Coutaz remain fully relevant in that they point to important distinctions regarding the structuring of an interactive application. For instance, there will be significant differences in the building of a software if one wants it to react, at runtime, to a change of the screen size rather than to recode it for each specific hardware target. However, in order to more accurately capture the range of possible reactive processes, we propose two additional extensions (Figure 1):

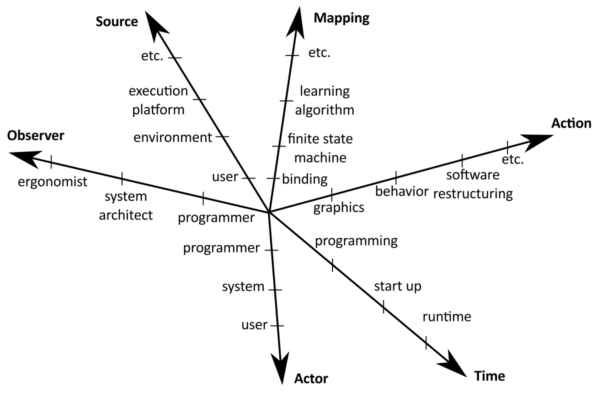

Figure 1. Analysis framework for reactive software processes, derived from Thevenin and Coutaz [36]

- refine the Target and Means axes into Source, Mapping and Action so as to denote what causes the process, what it triggers, and how the decision is made;
- enrich the Actor and Time axes to capture such cases as the redesign of a graphical component and the recompilation of an application after the notification of an API change.

The result is a design space with five axes describing what happens, how, when, whence and by whom, and one that characterizes whether this qualifies as an adaptation process.

Where the application programmer's work lie

Using our variant of Thevenin and Coutaz's analysis framework makes it easy to sort out what should be in the hands of application programmers. It suffices for that to identify which dimension in the design space should be assigned to whom. Not only does this help defining responsibilities for programmers and programming environment architects and ensuring the independence of their design choices. It also helps understanding what support programming primitives must offer in priority to the programmers of adaptive applications.

- The Observer dimension is not relevant to programming and can safely be ignored.
- What Sources and what Actions are used in a given application are the programmers' choice. However, defining the range of available sources and actions is the responsibility of the programming framework: does it provide support for multitouch devices, or only for classical pointing devices? Does it allow the design of rich graphical components, or only classical WIMP widgets? Consequently, these dimensions should be assigned to the framework programmer rather than the application programmer.
- Similarly, the Time and the Actor of the transformation strongly depend on the programming environment. Consider for instance the adaptation of an application to a new processor architecture. Currently, programmers must decide and implement this adaptation themselves because the application is unable to do it. In the future, this might become a feature of a programming environment that includes support for adaptation. As above, these dimensions concern more the framework programmer than the application programmer.
- The Mappings between sources and actions are where the intrinsic programming complexity resides. For any interactive behavior, the task of application programmers it to select the right mapping between sources and actions, or to build the appropriate mapping if necessary. This applies to adaptive software as well. All user interface frameworks support variants of "when this then that", and some offer state machines. For more complex cases, such as intelligent adaptation, programmers must build their own solutions from the basic mappings available. This is similar to using the control structures of standard programming languages to build dedicated algorithms, potentially very complex.

The proposed definition and analysis framework hence translate to the following hypothesis regarding application programmers: their task consists in using and creating mappings between events and transformations, therefore any programming environment that provides the appropriate primitives for creating mappings can be used without introducing dedicated primitives.

ADAPTATION IN A REACTIVE ARCHITECTURE

If the above hypothesis is valid, the primitive constructs of a classical reactive programming framework should be sufficient to address the whole range of adaptive processes delineated by our design space. We now test this hypothesis with the "event-control-action" schema from the reactive programming paradigm, refined into a an "event-control-transformation" schema. We show how this simple schema can be used to account for a series of classical adaptation scenarios.

Coupling sources and transformations

Simple (source, transformation) couples can be used to express a wide variety of behaviors:

- event sources, taken in a very broad sense as described in [11] ou [10], may be an input device consciously manipulated by a user, a physiological sensor, a physical sensor, a software component reporting the hardware configuration of the execution platform (displays, network, processor, etc.), and so on. The only requirement is that the programming environment provides these sources to the programmer so as to cover the desired part of the design space.
- Transformations can range from a simple variation of a graphical property to a complex restructuring of a component tree or to the loading of new software modules. Just as for the event sources, the limits to the expression of interactive processes are those of the actions made available by the programming environment.

With this schema in hand, we can easily describe some classic cases of adaptive processes. For instance, a Web service provider announces a change in their protocol through a dedicated service, application vendors develop an event source that catch such announcements and couple a rebuild action to it. Or a physiological sensor detects inappropriate attention patterns in a user, this is coupled to an animation that make the display vibrate so as break the attention pattern.

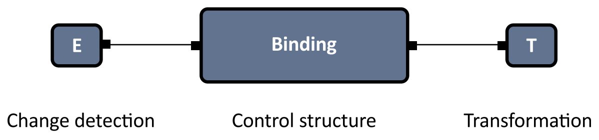

Figure 2. Simple binding between an event and a transformation

Continuous change

Another classic adaptive process is the adaptation to a continuous change in the application context. For example the continuous variation of ambient light, or more usual, the progressive resizing of the application window. In such cases we need a mapping that propagates the current value each time it changes, this is a typical dataflow mapping. However, the connected transformation does not need to be equally continuous, it may respond to various thresholds. In this case the mapping must be composed of a data flow block supplemented with a switch that will point to one branch or the other (figure 3).

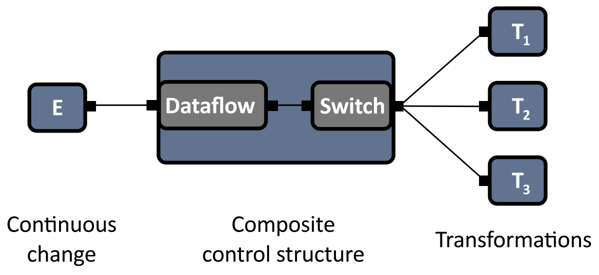

Figure 3. Composition of a dataflow and a switch

Dynamicity

The appearance and disappearance of an object in the surroundings of an application are classic sources of adaptive processes. Object is here taken in a very wide sense that includes input devices, users, software components, and so on. Such processes can be modeled by a component that encapsulates a monitoring process and that sends events when detecting that objects have appeared or disappeared. The appropriate control mapping is a simple binding that will trigger a specific transformation when receiving this event (figure 2). Of course the transformation itself may be complex, for example a transformation of the behavior of some interactors when a specific input device is plugged, however the mapping linking event and transformation is quite simple.

Complex algorithms

At first sight, artificial intelligence seems to offer a more challenging example of adaptation for our analysis schema. However it is no more the case if we regard it as a mapping of a higher degree of complexity. A neural network, for example, is an input/output structure that may be connected to one or several event sources on one side and to a set of transformations on the other side. The specificity of this mapping is that the link between events and transformations may change with time according to the change in the connection weights made by a learning algorithm.

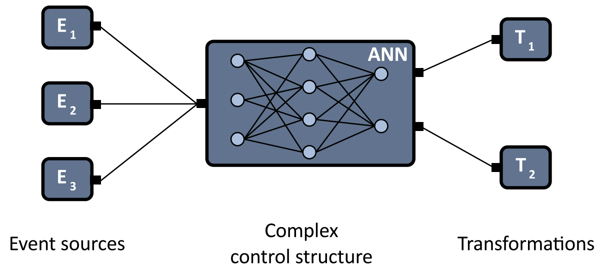

Figure 4. Learning algorithms as a complex mapping

Discussion

As we have seen, the proposed schema (event - control - transformation) provides an analysis tool that accounts for various classical adaptation scenarios. This provides initial validation not only of the schema, but also of part of the definition it is derived from: it makes sense to reduce adaptive behavior to interactive behavior as far as the programmer is concerned.

The above analysis also shows how the chosen schema helps to clarify in what cases programmers need to perform adaptation-specific programming tasks. Arguably, only the creation of dedicated control structures can be considered as such. This would be consistent with the classical distinction between adaptation in general and self-adaptive systems: the level of self-adaptation is probably correlated to the complexity of the algorithms involved in the control structures.

In order to further validate the proposed definition through the chosen programming schema, we have applied it to the implementation of an actual interactive software.

EXAMPLE APPLICATION

The djnn programming framework [1] implements the general event - control - action schema. We used it to implement a software prototype featuring various cases of interaction and adaptation, relying solely on the event - control - transformation principle.

A ground control station for UAVs

In the context of a research project dedicated to cooperation between humans and machines, we have developed a prototype ground control station for squad of civil unmanned aerial vehicles (UAV). This prototype is aimed at replacing the user interface of the open source Paparazzi system [7] in the future. Some of the requirements for this software are related to our concerns:

- the ground station must run on a classical desktop computer as well as on a tablet PC equipped with a touchscreen and stylus;
- it must be ready for multimodal interaction;
- the user interface must automatically adapt to changes in the operational configuration (network connectivity between the UAVs, current phase in the mission).

This makes it a concrete example of the issue we are discussing in this paper. We could have dealt with the requirements by choosing some conceptual frameworks from the literature, one for shared control and one for multimodality. But for our programmers, most concepts derived from these frameworks would have been irrelevant for the task. For example the question of who must take the control on the system and when is not their concern but those of the designer or the architect of the system. They are just interested in how to map a specific control source onto a specific set of transformations.

In the following, we illustrate how the principles described in the previous section can be used to build such mappings, which provide additional examples of the wide variety of mappings that the reactive architecture allows.

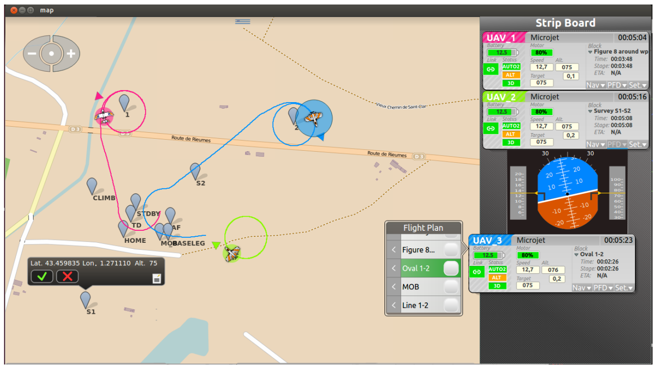

Figure 5. The main window of the UAV application

Application structure

This application is visually architectured in two main parts: a map picturing the flight area and a side panel containing various state indicators about each vehicle (figure 5). These indicators are packed in what is classically called a strip (figure 6). The flight plan of the UAVs are organized in series of waypoints that can be displayed and handled on the map. When the user selects a vehicle, various components appear and allow the user to interact with the UAV, by selecting a waypoint in its flight plan for example.

The application is implemented with the Java API of Djnn, a programming environment that implements the principles of reactive programming described earlier. In particular, Djnn comes with a series of control structures that all rely on the source-action principle and that correspond to the various mappings described in the previous section.

In addition, Djnn implements a model-based architecture in which the structure of an application is represented by components that can be created, assembled and destroyed in exactly the same way as the data manipulated by the application. This allows to trigger transformations of the application (e.g. the replacement of a visualization panel) with the same mechanisms as simple color or position changes. Consequently, all transformations of the software can be programmed according to the same source-action mappings, provided that the necessary event sources are available.

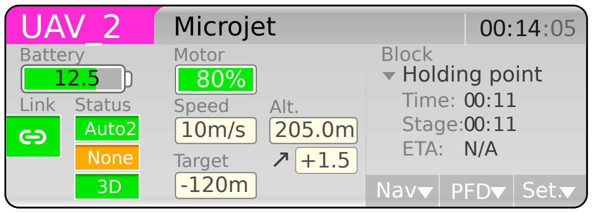

Figure 6. "Strip" gathering the various state information of a vehicle

Event sources

The application is sensitive to events from the UAV squad as well as user events.

UAVs are equipped with autonomous squad flight capabilities and the communication between the UAVs and the ground station relies on a dynamic routing protocol. Changes in the squad configuration or in the routing configuration, as well as alarms from UAVs (e.g. "short fuel") are transmitted to the application and must trigger reconfigurations of the user interface. For instance, in case of an emergency the application should restructure the interface so as to force the operator to focus on a UAV, with all the relevant information easily available. A first implementation step has consisted to make all these messages from the UAVs and routers available to the rest of the application in the same form as input events.

The user events are the usual ones: mouse clicks, touch panel touches, etc. A future version of the application includes unconscious input, provided by physiological sensors (EEG, near infra red). For this reason, and because the application must run on various platforms, the application must be able to transform its internal wiring according to the available devices. As above, this is possible because events such as the connection and disconnection of input devices are available to the rest of the application.

In the following we give an overview of three kinds of coupling between events and transformations of the application. All are implemented according to very similar patterns, with the same mappings. From the user's point of view, some of these couplings will appear as simple interaction, others as adaptive processes, others with no clear status. The homogeneous implementation model allows to explore design spaces where the difference between adaptation and interaction is not important.

Basic interaction

Like most of the graphical user interfaces, our application proposes many examples of simple interactions such as state change on mouse press or graphical object dragging. This is the case for the waypoints, which can be in four different states (figure 7). The transitions between the states are triggered by the user's actions on a mouse or a touch screen.

The coupling between events and changes is described with a finite state machine that governs the branches of a switch as explained in [11]. Each transition corresponds to a coupling, where the event is for instance a mouse press or a screen touch and the transformation is the activation or the deactivation of a graphical object.

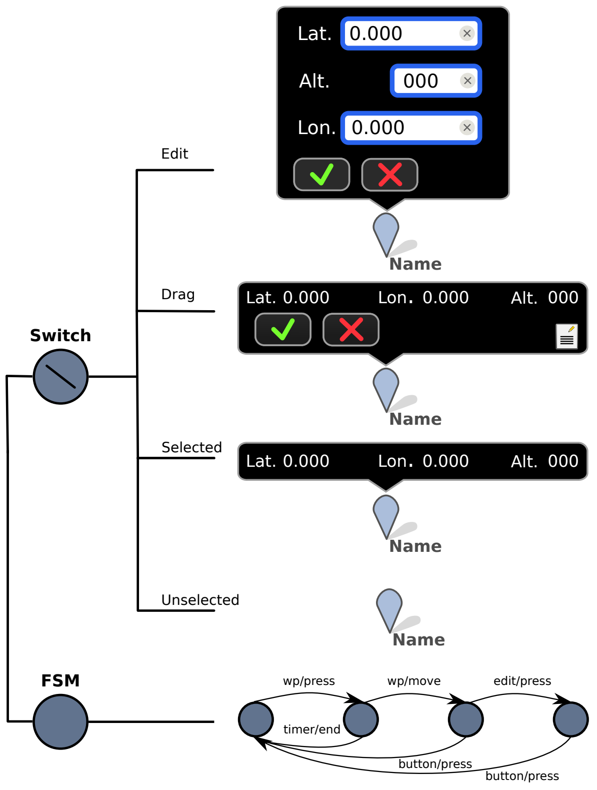

Figure 7. Waypoint state change triggered by a user action

Recomposing the visual architecture

The reorganization of the visual architecture is a classic case of adaptation [35]. In its current version, our application offers two kinds of such a reorganization. The first one is triggered by an alarm coming from the anti-collision system of the UAVs. The event is a message received on a software bus, that contains the ID of the concerned vehicles as well as the avoidance strategy they have adopted. In our application, this message is encapsulated in a *TCAS-alert* component that represents the various properties of the alert. When a message arrives, this component is activated. This activation is bound to the activation of visual components that display the alert parameters. Consequently, the user's current activity is interrupted by the disappearance of all the usual displays (flight plan, altitude controller, etc.), that are replaced by the altitude of the vehicles involved, their slope and the avoidance strategy.

This transformation implements a new distribution of authority. It is aimed at improving the awareness of operators and preventing them from further actions on the flight parameters that could jeopardize the automatic avoidance strategy. While this can be described as a typical scenario of adaptation, the event mappings involved in this process are the same as in the previous case: a subscription to an event *TCAS-alert* that activates a few components.

Another classic case of graphical reorganization is the one motivated by a change in the dimensions of a container (window, panel, widget, etc.). The strip panel offers two levels of

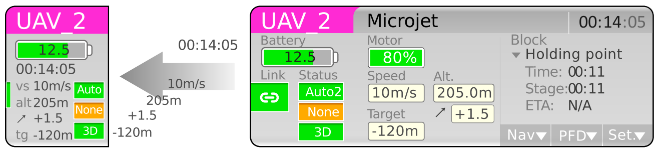

Figure 8. Animated strip adaptation, or manual strip resizing

graphical reorganization. The first is a simple size reduction of the panel itself when the main display becomes too small. The second level is a complete reorganization of the strips themselves when the panel becomes too small. This can happen when the application is run on a smaller display, when the main window is resized or when the user decides to reduce the size of the panel. In addition, when the user acts on the panel she directly takes the control over the animation of a smooth graphical transformation inspired from [15] (figure 8).

This transformation cannot be described by a simple finite state machine because of its progressiveness. The mapping involved here is a combination of dataflow components and finite state machines. The interesting point is that the same transformation can be triggered and controlled in two different ways, either automatically on window resizing or manually on direct user action. In other words the same component, driven by the same mapping, can be triggered in both a way that is pure interaction and a way that is a fairly complex case of adaptation.

Adapting interaction styles

The third kind of coupling proposed by our application concerns the modification of the behavior of some graphical components according to the hardware architecture and the available input devices. The transition from a classic computer equipped with a mouse and a keyboard to a tablet PC with a touchscreen raises several known problems. One of them is the fact that a finger on a touchscreen can hide a too small target, leaving the user without feedback. The usual solution [25] that we have adopted consists in the addition of a deported feedback over small components (figure 9). The other issue comes from the fact that most touchscreens have no hovering events [8]. It is thus necessary to transform the state machine that governs the buttons according to the current input device.

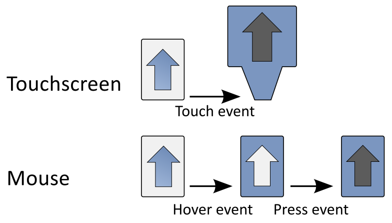

Figure 9. Dual behavior of the buttons

Here, we have two kinds of transformation, a graphical one and a behavioral one, both triggered by appearing/disappearing of an input device. The corresponding adaptation is obtained by coupling these transformations to events such as "new touchscreen available" and "new mouse available". For simple hardware configurations (one mouse or one touch-screen), it is enough to group these couplings in a finite state machine, which is not much different from that of section "Basic interaction" above. The only significant difference is the nature of the events that trigger the transitions.

This series examples demonstrate that if some situations can be distinguished from the user's point of view, or from the observer of human-computer interaction, from the programmer's point of view they can be analyzed according to the same canonical schema. Furthermore, some combinations can be imagined that defeat all attempts at strict categorization of software behavior.

CONCLUSION

In this article, we have addressed support for adaptation under a new angle: the point of view of programmers of interactive software. We have untangled the concerns from software architecture and those from adaptive systems so as to propose two independent albeit compatible models.

On one hand we came up with a definition and a design space for software adaptivity that is grounded in the more general definition of system adaptivity. On the other hand creating adaptive software comes through as a straightforward application of the reactive programming model. In simple cases, adaptation can managed with usual software patterns for interactivity. In more complex cases, control patterns are required but the reactive model still holds.

Not only does this offer some clarifications on software adaptivity, it also opens new opportunities for designing control structures for elaborate adaptive behaviors. This paves the way to innovative combinations of interaction and adaptation, and to a better integration of interactive software and intelligent systems.

ACKNOWLEDGEMENTS

This work was funded by the ARTEMIS JU and the French government through project D3CoS. The authors are grateful to Stéphane Conversy and to the reviewers of a previous version of this article for their constructive comments.

REFERENCES

1. `http://djnn.net`.
2. Aubin, J.-P., Bayen, A., Bonneuil, N., and Saint-Pierre, P. *Viability, Control and Games: Regulation of complex evolutionary systems under uncertainty and viability constraints*. Springer, 2005.
3. Baldauf, M., Dustdar, S., and Rosenberg, F. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing 2*, 4 (2007), 263–277.
4. Bass, L., Pellegrino, R., Reed, S., Seacord, R., Sheppard, R., and Szezur, M. R. The Arch model: Seeheim revisited. Presented at the CHI'91 User Interface Developers Workshop, Apr. 1991.
5. Ben Mahmoud, M., Larrieu, N., Pirovano, A., and Varet, A. An adaptive security architecture for future aircraft communications. In *Proceedings of IEEE/AIAA DASC 2010* (2010), 3.E.2–1–3.E.2–16.
6. Bencomo, N., and Blair, G. Using architecture models to support the generation and operation of component-based adaptive systems. In *Software Engineering for Self-Adaptive Systems, LNCS 5525*. Springer, 2009.
7. Brisset, P., and Hattenberger, G. Multi-UAV control with the paparazzi system. In *Conference on Human Operating Unmanned Systems* (2008).
8. Buxton, W. A. S. A three-state model of graphical input. In *Proceedings of INTERACT'90*, Elsevier (1990).
9. Calvary, G., Serna, A., Kolski, C., and Coutaz, J. *Transport: a fertile ground for the plasticity of user interfaces*. ISTE Ltd and John Wiley & Sons, Inc., 2011, 343–368.
10. Chatty, S., Lemort, A., and Valès, S. Multiple input support in the IntuiKit framework. In *Proceedings of Tabletop 2007*, IEEE computer society (2007).
11. Chatty, S., Sire, S., Vinot, J., Lecoanet, P., Mertz, C., and Lemort, A. Revisiting visual interface programming: Creating GUI tools for designers and programmers. In *Proceedings of UIST'04*, Addison-Wesley (Oct. 2004), 267–276.
12. Chen, H., Finin, T., and Joshi, A. An ontology for context-aware pervasive computing environments. *The Knowledge Engineering Review 18*, 3 (2003), 197–207.
13. Collignon, B., Vanderdonckt, J., and Calvary, G. Model-driven engineering of multi-target plastic user interfaces. In *Proceedings of ICAS 2008*, IEEE Computer Society Press (2008), 7–14.
14. Dey, A. K., Abowd, G. D., and Salber, D. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction Journal 16*, 2 (2001), 97–166.
15. Dragicevic, P., Chatty, S., Thevenin, D., and Vinot, J.-L. Artistic resizing: A technique for rich scale-sensitive vector graphics. In *Proceedings of UIST'05*, ACM (2005), 201–210.
16. Dragicevic, P., and Fekete, J.-D. The input configurator toolkit: Towards high input adaptability in interactive applications. In *Proceedings of AVI'04*, ACM (2004), 244–247.
17. Gajos, K., and Weld, D. S. Supple: Automatically generating user interfaces. In *Proceedings of IUI'04*, ACM (2004), 93–100.
18. Greenberg, S., and Witten, I. H. Adaptive personalized interfaces - a question of viability. *Behaviour and Information Technology 4*, 1 (1985), 31–45.
19. Gu, T., Pung, H. K., and Zhang, D. Q. A service-oriented middleware for building context-aware services. *Journal of Network and Computer Applications 28*, 1 (2005), 1–18.

20. Healey, J., and Picard, R. Detecting stress during real-world driving tasks using physiological sensors. *IEEE Transactions on Intelligent Transportation Systems 6*, 2 (2005), 155–166.
21. Henricksen, K., and Indulska, J. Developing context-aware pervasive computing applications: Models and approach. *Pervasive and Mobile Computing 2*, 1 (2006), 37 – 64.
22. Holland, J. H. *Adaptation in natural and artificial systems*. MIT Press, 1975.
23. Innocent, P. Towards self-adaptive interface systems. *International Journal of Man-Machine Studies 16*, 3 (1982), 287 – 299.
24. Kiczales, G. Aspect-oriented programming. *ACM Computing Surveys 28*, 4es (1996).
25. Martin, B., Isokoski, P., Jayet, F., and Schang, T. Performance of finger-operated soft keyboard with and without offset zoom on the pressed key. In *Proceedings of Mobility'09*, ACM (2009), 59:1–8.
26. McKinley, P., Sadjadi, S., Kasten, E., and Cheng, B. H. C. Composing adaptive software. *Computer 37*, 7 (2004), 56–64.
27. McTear, M. User modelling for adaptive computer systems: a survey of recent developments. *Artificial Intelligence Review 7*, 3-4 (1993), 157–184.
28. Nilsson, E. G., Floch, J., Hallsteinsen, S., and Stav, E. Model-based user interface adaptation. *Computers and Graphics 30*, 5 (2006), 692–701.
29. Norcio, A. F., and Stanley, J. Adaptive human-computer interfaces: A literature survey and perspective. *IEEE Transactions on Systems, Man, and Cybernetics 19*, 2 (1989), 399–408.
30. Papazoglou, M., Traverso, P., Dustdar, S., and Leymann, F. Service-oriented computing: State of the art and research challenges. *Computer 40*, 11 (2007), 38–45.
31. Parasuraman, R. Neuroergonomics: Research and practice. *Theoretical Issues in Ergonomics Science 4*, 1-2 (2003), 5–20.
32. Pfaff, G. E., Ed. *User Interface Management Systems*. Eurographics Seminars. Springer, 1985.
33. Schmidt, A., Beigl, M., and Gellersen, H.-W. There is more to context than location. *Computers and Graphics 23*, 6 (1999), 893 – 901.
34. Solovey, E. T., Girouard, A., Chauncey, K., Hirshfield, L. M., Sassaroli, A., Zheng, F., Fantini, S., and Jacob, R. J. Using fNIRS brain sensing in realistic HCI settings: experiments and guidelines. In *Proceedings of UIST'09*, ACM (2009), 157–166.
35. Stuerzlinger, W., Chapuis, O., Phillips, D., and Roussel, N. User interface façades: Towards fully adaptable user interfaces. In *Proceedings of UIST'06*, ACM (2006), 309–318.
36. Thevenin, D., and Coutaz, J. Plasticity of user interfaces: Framework and research agenda. In *Proceedings of INTERACT'99*, IOS Press (1999).
37. Wang, H., Prendinger, H., and Igarashi, T. Communicating emotions in online chat using physiological sensors and animated text. In *CHI '04 Extended Abstracts*, ACM (2004), 1171–1174.
38. Wolpaw, J. R., Birbaumer, N., McFarland, D. J., Pfurtschellere, G., and Vaughan, T. M. Brain-computer interfaces for communication and control. *Clinical Neurophysiology*, 113 (2002), 767–791.