Modular Design of Man-Machine Interfaces with Larissa

K. Altisen, F. Maraninchi, and D. Stauch

Verimag, Centre équation - 2, avenue de Vignate, 38610 GIÈRES — France

Abstract. The man-machine interface of a small electronic device like a wristwatch is a crucial component, as more and more functions have to be controlled using a small set of buttons. We propose to use Argos, an automaton-based language for reactive systems, and Larissa, its aspect-oriented extension, to show that several interfaces can be obtained from the same set of basic components, assembled in various ways. This is the basis of a quite general component-based development method for man-machine interfaces.

1 Introduction

Man-Machine Interfaces of small electronic devices. In small devices such as wristwatches, portable multimedia devices, or GPS devices, more and more functions have to be controlled using a very small set of buttons. The design of such systems usually follows an approach in which the interface is clearly separated: this is a component that accepts the button events as inputs, and translates them into complex functions, depending on its internal state, or mode. For instance, the same button of a wristwatch means "toggle alarm" or "increment minutes", depending on the running mode. We will distinguish the *interface* from the *internal* components of the system, which take a much larger set of inputs ("toggle alarm", "increment minutes", etc.). In this paper, we concentrate on the design of the *interface* component of such small electronic devices. We propose to use aspects in order to help modular design and reuse.

Programming Man-Machine Interfaces. Man-machine interfaces are typical interactive, or reactive systems. Using reactive languages for programming or modeling them is quite natural. Moreover, among the formalisms and languages that are used to describe reactive systems, those that are based on explicit automata, like Statecharts [8], are particularly well adapted. The user documentation of a small electronic device is often given with partial graphical automata, because it is the more natural way of thinking of it.

The family of *synchronous languages* [5] has been very successful in offering semantically founded languages, adapted to the needs of the programmers. It is comprised of several dataflow languages (Lustre, Signal), a textual imperative language (Esterel), and several variants of graphical automaton-based languages (Argos, Safe State Machines, some variants of Statecharts). Their main structure is the parallel composition; components synchronize and exchange data via

the so-called synchronous broadcast. All these languages may be compiled into sequential cyclic code for a direct implementation on embedded processors, or into synchronous circuits, for ASIC or FPGA implementations. Moreover, the internal structure used in the compilation or synthesis process can be used as input by various formal verification tools (model-checkers, abstract-interpretation tools, theorem-proving tools).

In this paper, we use a simple version of Argos [14]. Argos was first designed as a variant of Statecharts having a pure synchronous semantics. The hierarchy of states inherited from Statecharts is very convenient for the description of a watch interface that has several running modes.

Aspect Oriented Programming. Programming languages usually have mechanisms to *structure* the code of programs. For any program P, whatever be its modular structure, it is always possible to think of some functionality F that P should provide, in such a way that F cannot be implemented only by some modular modifications of P by e.g. simply adding some new components to it. This is the case when F is a *crosscutting* feature, that indeeds requires modifications in several components of the original program. Aspect oriented programming (AOP) has emerged recently as a response to this problem. It provides facilities to design F as a new kind of component – F is then called an *aspect* – and to compile F and P together – this process is called the *weaving* of the aspect F into the program P. Some aspect languages like AspectJ [9] are becoming increasingly popular. A number of case studies (see for instance [2, 3]) have shown that they can considerably improve the code structure of large systems.

Aspects may be used to describe functionalities like tracing, debugging, profiling. In this case, they do not aim at modifying the behavior of the original program; they only add code that *observes* it. They should be given a complete view of the entities of the program. On the other hand, aspects may sometimes be used in order to modify the behavior of the original program. For this kind of aspects, the way they modify the program has to be clearly defined. Aspect weaving can also be required to have some good properties, like the preservation of a behavior equivalence (if two programs P and Q behave the same, then the result of weaving an aspect A into P should behave the same as the result of weaving the same A into Q). In this paper, we use the formally defined language Argos, and its aspect extension Larissa [1]. The aspects allowed have a clear semantics and the weaving process preserves the usual behavior equivalence.

Contributions and structure of the Paper. We show the interest of using aspect oriented programming in the particular context of developing man-machine interfaces of small electronic devices: we illustrate it by studying several variants of watches. We propose an approach in which these interface components may be described by assembling smaller components, thus improving reuse.

We consider aspects as components in this assembling process. This is made possible by two important points: first, our mechanism for aspect weaving behaves exactly as the operators of the base language, and has the same properties regarding the respect of a behavior equivalence; this allows to combine pieces of

programs and aspects freely, in any order. Second, our specification of aspects is independent of the internal structure and names of the base program, and only refers to the elements of its interface; this means that a component on which an aspect is applied may be safely replaced by another one, provided its interface is the same and its behavior is equivalent to the old one.

The structure of the paper is as follows: Section 2 describes the base language Argos; Section 3 describes our aspect extension of Argos, from the user point of view; Section 4 is the case study; Section 5 comments the case study; Section 6 is a non-exhaustive list of related work and Section 7 concludes and lists the main perspectives.

2 The Argos language

An Argos program describes the *reactive kernel* of a system. A reactive system is a computer system that communicates with the environment it is embedded in: it has input signals coming from the environment and output signal it emits towards the environment. In Argos, input and output signals have Boolean values. Whereas the environment evolves in a continuous manner, the fact that the reactive system is a program implies that, from the program(mer)'s point of view, the time is sampled into instant. At each instant, an Argos program reacts to inputs by sending outputs and updating its internal memory. Such a reaction is *atomic*: the system does not read inputs while computing outputs and updating its memory. This property mainly characterizes synchronous languages of which Argos is a member.

Argos is an automata based language. Its base components are automata with transitions labelled by inputs and outputs; more complex components can be obtained by connecting components with operators, i.e. the parallel product between automata, the encapsulation (hiding variables), the inhibition (freezing a program for a while) and the hierarchy (a state of an automaton may contain a program). The communication between components is achieved by parallel product and encapsulation. Two programs communicate by exchanging local signals which are inputs of one program and outputs of the other. The communication is the *synchronous broadcast*: it is non blocking (unlike the *rendez-vous* mechanism, for instance).

Argos programs are (as programs should always be) deterministic and complete, i.e. for any given sequence of inputs there exists a unique execution of the program. The semantics of Argos is formally defined by using traces of the execution. Those traces are only defined by the values of the inputs and outputs at each instant (the states reached – value of the memory – are not part of the information of a trace). A semantic equivalence between programs is also defined as being the equality of traces.

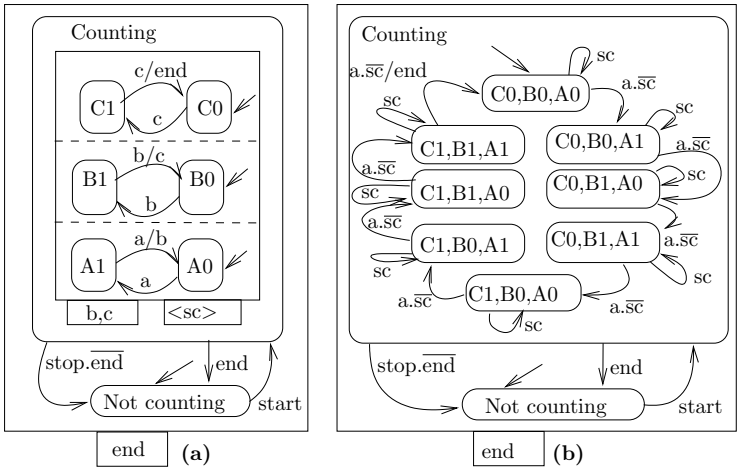


Fig. 1. Two Argos programs for the modulo-8 `a`-counter

2.1 Syntax of the main constructs

The complete language is described in [14]. In this paper, we partially describe it using an example. Figure 1(a) is an Argos program using four automata to describe a modulo-8 `a`-counter.

Single automata. Rounded-corner boxes are automaton states; arrows are transitions. A set of states and transitions which are connected together constitutes an automaton. The four basic components of the program have the following sets of states: {`Counting, Not counting`}, {`A0, A1`}, {`B0, B1`}, {`C0, C1`}. Transitions are labeled by a Boolean condition on input signals, and a set of emitted signals. We use the concrete syntax: `condition / emitted signals`. In the condition, negation is denoted by overlining and conjunction is denoted by a dot (examples: `c/end`, `stop.`$\overline{\texttt{end}}$). When the output set is empty, it can be omitted. The initial state is designated by an arrow without source. States are named, but names should be considered as comments: they cannot be referred to in other components nor are used to define the semantics of the program. An arrow can have several labels — and stand for several transitions, in which case the labels are separated by a comma. By convention, every automaton is complete: if a state has no transition for some input valuations, we suppose that there is a self-loop transition with these valuations as triggering condition and no outputs.

State refinement. The automaton whose states are `Counting` and `Not counting` is said to be *refined*. The `Counting` state contains a sub-program built from the three other automata.

Parallel product. Three automata whose states are respectively {`A0, A1`}, {`B0, B1`}, {`C0, C1`} are put *in parallel*: they are drawn separated by dashed lines.

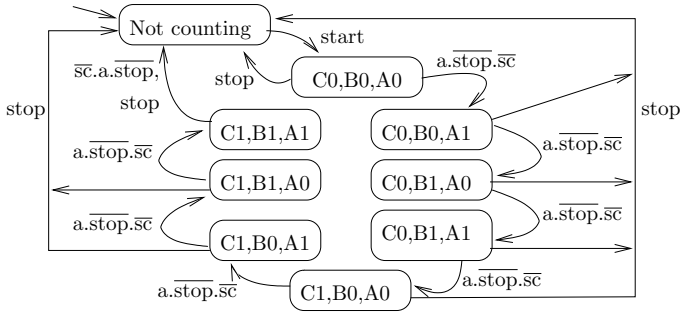


Fig. 2. The behavior of the modulo-8 `a`-counter

Encapsulation. Rectangular boxes are used for unary operators. The external box, whose cartridge contains `end`, is the graphical syntax for the declaration of a local signal `end`. The box defines the scope in which `end` is known. This signal is used as input by the refined automaton; it is used as output by one of the three other ones: a communication will take place between the two. The same operator is used in order to limit the scope of signals `b, c` to the program constituted by the three unrefined automata.

Inhibition. The inhibition is another unary operator: the notation is another cartridge containing a *fresh* variable between "<" and ">". The parallel product of the three non refined automata are inhibited by the inhibition variable `sc`.

Interface of a program. All signals which appear in a left-hand (resp. right-hand) side of a label plus inhibition variables, and are not declared to be local to some part of the program are *global inputs* (resp. *global outputs*).

2.2 Intuitive semantics

We give here the intuitive semantics of the operators, by explaining the behavior of the counter. This behavior is a single automaton, as shown by Figure 2.

First, observe the three automata embedded in a parallel structure, and the operator which defines the scope of `b` and `c`. This constitutes a subprogram whose only input is `a`, and whose only output is `end`. The global behavior of this subprogram is defined by: the global initial state is `C0,B0,A0`; when it has reacted to input `a` n times, the program is in state $\mathtt{C}_k\mathtt{,B}_j\mathtt{,A}_i$, where $i+2j+4k = n \bmod 8$; `end` is emitted every 8 `a`'s.

This behavior is achieved by connecting three one-bit counters. The first one (A) reacts to external input `a`, and triggers the second one (B) with signal `b`, every two `a`'s. The second one, reacting to `b`, triggers the third one (C) with `c`, every two `b`'s. The third one emits `end` every two `c`'s. The communication being *synchronous*, a reaction to which the three bits participate, is indeed *one* transition in the global behavior (e.g., reaction to `a` from `C0,B1,A1` to `C1,B0,A0`). Finally, inhibition by `sc` (for "stop counting") is applied: in each state, either `sc` is true and the automaton stays in its current state, or it is false and the

automaton behaves as before applying inhibition. The result of those operators is shown in Figure 1(b): the modulo-8 counter subprogram modularly described in Figure 1(a) has been replaced by an equivalent 8-state automaton.

Refining the `Counting` state with the modulo-8 counter subprogram provides a way to describe how the counter can be started and stopped. Provided that `start` is true, the transition which enters the `Counting` state always goes to its initial state `C0,B0,A0`. The counter reacts to the occurrences of `a` and `sc`; and the refined automaton reacts to the occurrences of `stop` and `end`. The `Counting` state is left if `end` and/or `stop` occurs. At the instant `Counting` is left, the refining subprogram still reacts. Thus, the modulo-8 counter can terminate itself by emitting `end`. The program goes to state `Not counting` when `stop` occurs at any time in state `Counting`, and also when it is in state `C1,B1,A1` and `a` and $\overline{\texttt{sc}}$ occur, because `end` is emitted. `end` is also emitted when `stop`, `a` and $\overline{\texttt{sc}}$ occur together in state `C1,B1,A1`.

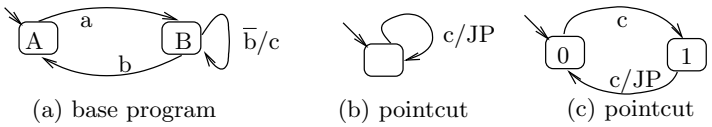
As we did for the example, the semantics of each Argos operator is given by a flattening operation that transforms any complex program (made of automata composed together) into an equivalent single flat automaton. The semantics of a flat automaton is then given by defining the set of all its execution traces, (inputs and outputs at each instant).

3 Larissa: an Aspect Extension to Argos

Argos operators are already powerful. However, there are cases in which they are not sufficient to modularize all concerns of a program: some small modifications of the global program's behavior may require that we modify all parallel components, in a way that is not expressible with the existing operators.

The goal of aspects being precisely to specify some cross-cutting modifications of a program, we proposed an aspect-oriented extension for Argos [1], which allows the modularization of a number of recurrent problems in reactive programs, like the reinitialization. This leads to the definition of a new kind of operators (corresponding to the weaving of aspects) for which we took care of ensuring some nice properties: they preserve determinism and completeness of programs and also the semantic equivalence between programs.

All the aspect extensions of existing languages (like AspectJ [9]) seem to share two notions: pointcuts and advice. The *pointcut* describes a general property of program points where a modification is needed (all the methods of the class `X`, all the methods whose name contains `visit`, etc.); the pointcut, applied to a particular program, selects a set of concrete *join points*, where the aspect has to be applied. The *advice* specifies what has to be done at each of these join points (execute some piece of code before the normal code of the method, for instance). For Larissa, we adopted this approach: an aspect is given by the specification of its pointcut and its advice.



(a) base program (b) pointcut (c) pointcut

Fig. 3. Example pointcuts

3.1 Join Point Selection

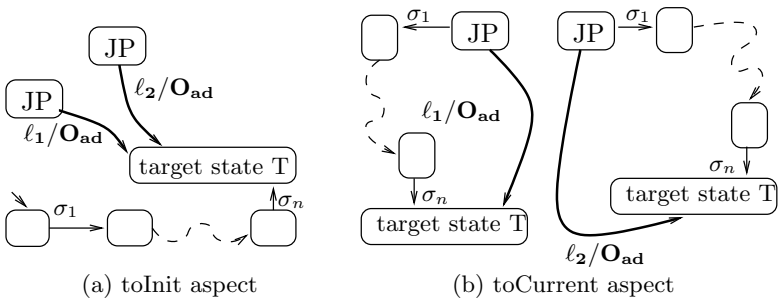
In Larissa, we decided not to express pointcuts in terms of the internal structure of the base program. For instance, we do not allow pointcuts to refer explicitly to some state name (as AspectJ can refer to the name of a private method). As a consequence, pointcuts may refer to the observable behavior of a program only, i.e., its inputs and outputs. In the family of synchronous languages, where the communication between parallel components is the synchronous broadcast, *observers* [7] are a powerful and well-understood mechanism which may be used to describe pointcuts. Indeed, an observer is a program that may observe the inputs and the outputs of the base program, without modifying its behavior, and compute some safety property (in the sense of safety/liveness properties as defined in [10]).

In Larissa, pointcuts are expressed as observers, which select a set of *join point transitions* by emitting a single output `JP`, the *join point signal*. A transition T in a program P is selected as a join point transition when in the concurrent execution of P and the pointcut, `JP` is emitted when T is taken. Technically, we perform a parallel product between the program and the pointcut and select those transitions in the product which emit `JP`. Figure 3 illustrates the pointcut mechanism. The pointcut (b) specifies any transition which emits `c`: in base program (a), the loop transition in state B of the base program is selected as a join point transition. The pointcut (c) specifies every second time `c` is true: no transition of the base program (a) corresponds directly to this condition. However, as the join points are selected on the parallel product of the base program and the pointcut, the pointcut introduces new memory: the automaton memorizes if `c` has been emitted an even or an odd number of times.

Pointcuts can be built by composing other pointcuts with Argos operators. E.g., some pointcuts can be put in parallel with an automaton which takes their join point signals as inputs and emits the join point signal of the composed pointcut. Thus, expressions like "pointcut A and not pointcut B" or "pointcut A until `a` and then pointcut B" can be written modularly.

3.2 Specifying the advice

The advice usually expresses the modification applied to the base program. In our setting, we consider that the base program has been flattened first, as explained in Section 2.2. In Larissa, we defined two types of advice: in the first type, an advice replaces the join point transitions with *advice transitions* pointing to some



(a) toInit aspect (b) toCurrent aspect

Fig. 4. Schematic toInit and toCurrent aspects. (Advice transitions are in bold.)

existing target states; in the second type, an advice introduces a full program between the source state of the join point transition and some existing target state. In both cases, target states have to be specified without referring explicitly to state names.

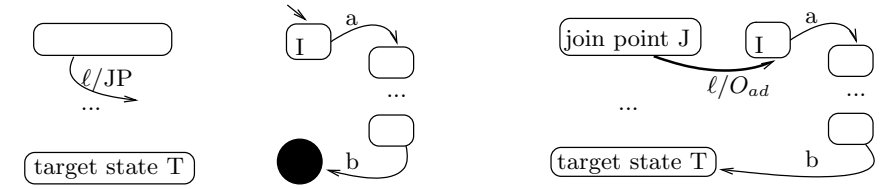
We consider three ways of specifying the target state T, among the existing states of the base program P: 1) T is the state of P that would be reached by executing some finite input trace from the initial state of P, called a *toInit* advice; 2) T is the state of P that would be reached by executing some finite input trace from the join point itself, called a *toCurrent* advice; 3) we first define some recovery states, among the states of P; then T is the recovery state that was passed last. The third type will not be used in the paper (see [1] for further details). For the first two types, specifying the advice includes giving a finite input trace to define the target state. Since the base program is both deterministic and complete, executing an input trace from any of its states is an effective way of defining exactly *one* state.

Advice Transition. The first type of advice consists in replacing each join point transition with an advice transition. Once the target state is specified by a finite input trace $\sigma = \sigma_1 \dots \sigma_n$, the only missing information is the label of these new transitions. We do not change the input part of the label, so as to keep the woven automaton deterministic and complete, but we replace the output part by some *advice outputs* O_{ad}. These are the same for every advice transition, and are thus specified in the aspect. Advice transitions are illustrated in Figure 4.

Advice Program. It is sometimes not sufficient to modify single transitions, i.e. to jump to another location in the automaton in only one step. It may be necessary to execute arbitrary code when an aspect is activated. In these cases, we can insert an automaton between the join point and the target state.

Therefore, we use an *inserted automaton* A_{ins} that *terminates*. Since Argos has no built-in notion of termination, the programmer of the aspect has to identify a final state `F` (denoted by filled black circles in the figures).

Inserting an automaton is quite similar to inserting a transition. We first specify a target state T by a finite input trace, starting either from the initial



(a) base program (b) inserted automaton A_{ins} (c) the woven program

Fig. 5. Inserting an advice automaton

state or from the source state of the join point transition. Then, for every T, a copy of the automaton A_{ins} is inserted, which means: 1) replace every join point transition J with target state T by a transition to the initial state I of this instance of A_{ins}. As for advice transitions, the input part of the label is unchanged and the output part is replaced by the *advice outputs* O_{ad}; 2) connect the transitions that went to the final state `F` in A_{ins} to T. See Figure 5.

3.3 Fully Specifying an Aspect

As stated above, an aspect is given by the specification of its pointcut and its advice: $Asp = (\texttt{PC-program}, Advice)$. `PC-program` is an Argos program with a single output `JP` used as the pointcut program. *Advice* is a tuple which contains 1) the advice outputs O_{ad}; 2) the *type* of the target state specification (*toInit* or *toCurrent*); 3) the finite trace σ over the inputs of the program; and optionally 4) when adding an advice program, `ADV-program`, the advice program itself (when adding advice transitions, this slot is left empty).

As a summary, when adding an advice transition, $Advice = < O_{ad}, type, \sigma >$, when adding an advice program, $Advice = < O_{ad}, type, \sigma,$ `ADV-program`$>$, with $type \in \{toCurrent, toInit\}$.

3.4 Formal Setting and Implementation

In [1], we define the aspect language formally, and prove the main properties: aspect weaving preserves the usual behavior equivalence, and also preserves the determinism and completeness of the base program. Let us note $P \triangleleft Asp$ the result of weaving an aspect Asp into a program P.

The preservation of the equivalence, noted $\sim$, means that, if $P \sim Q$ then, for any Asp, $(P \triangleleft Asp) \sim (Q \triangleleft Asp)$. With these properties, aspect weaving can indeed be considered as a new operator. This new operator can be used freely in expressions of the form: $((P||(Q \triangleleft A_1))||R) \triangleleft A_2$, for instance ($||$ denotes the parallel composition). Then, any of the components appearing in this expression may be replaced by an equivalent one without changing the behavior of the global program.

A compiler [11] for Larissa was developed, as an extension of an existing Argos compiler: it performs the weaving of an aspect Asp into an Argos program P

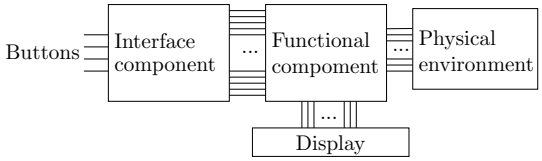


Fig. 6. Typical structure of a small electronic device.

as shown above. This tool is connected to simulation, test, debug and formal verification tools like the model-checker Lesar [6].

The formal definition of aspects also allows to study interference problems in a clean setting. This question is treated in [16].

4 Case Study: a Suunto[1] Watch

4.1 Global Scheme

In this section, we model the interfaces of small electronic devices with Argos and Larissa. These devices – e.g. wristwatches, alarm clocks or car radios – usually have a small number of buttons which control a large number of functionalities. These buttons have different meanings depending on the state in which the device is currently.

Therefore, controllers of such devices usually have a structure like the one shown in Figure 6: it contains an *interface component*, which interprets the meaning of the buttons the user presses, and then calls the corresponding function in the underlying *functional component*. The functional component obtains the necessary information of the environment of the device (e.g. a quartz crystal to measure time), reads and writes persistent memory, and updates the display.

Hierarchic automata languages like Argos are very well suited to model interface components. However, some additional functions are difficult to express in a modular way.

Our case study shows two of these functions: shortcuts, and additional modes. A shortcut is the possibility, in some given modes, to use a single button to activate a function that would otherwise need a long sequence of buttons. Adding shortcuts modifies the interface, but not the internal components.

Furthermore, interfaces for similar devices often use the same components for large parts of their functionalities. We show that aspects can be used to compose and configure components so that the same components can be used for different devices.

4.2 Suunto[1] Watches

As a case study, we implement the interface components of two complex wristwatches, the Altimax[1] and the Vector[1] models by Suunto[1]. Both share the same

[1] Suunto, Altimax and Vector are trademarks of Suunto Oy.

casing, display, and a large set of their functionalities: time, altimeter and barometer functions are nearly equal in both models, but the Vector also has an integrated compass. Carefully following the documentation, we propose Argos components and aspects to describe the interfaces of the two watches.

The Base Program. In both watches, each main functionality is represented by a main mode, which in turn has several submodes, that offer numerous functionalities. The interfaces of both watches contain four buttons, the `Mode`, the `Select`, the `Plus`, and the `Minus` button. The `Mode` button circles between the main modes, or, in a submode, returns to the main mode. The `Select` button selects a submode, and the `Plus` and `Minus` buttons modify current values. All main modes and many submodes have an associated configuration mode, where settings for the mode can be modified. A configuration mode can be reached by holding the select button pressed for two seconds in the corresponding mode.

Figure 7 shows the implementation of the interface component for the modes both wristwatches have in common. The input `s2s` occurs when the `Select` button is pressed for two seconds. This part of the interface is called the *base program*. Figure 7 is not complete: most of the states are further refined, and only some of the outputs (i.e. the commands to the functional component) are shown, namely `Time-Mode, Bari-Mode, Alti-Mode` and `mainMode`. The signal `toMainMode` is encapsulated: the submodes can emit it to force a return to their main mode. To save space, the encapsulation is not included in Figure 7.

The Fast Cumulative Shortcut. The altimeter in the watches can record vertical movements in so called *logbooks*, so that the user can evaluate his performance after a hike. A logbook records the distances the user vertically ascends and descends from the moment it is started until it is stopped, and the number of runs accomplished in this period, i.e. the vertical movements of at least 50 meters. However, a logbook can only be read after recording stopped, and it is quite complicated to display the logbook (one has to go to the third submode of the altimeter main mode). Therefore, the Altimax model has the *fast cumulative shortcut*: in any main mode, when the `Minus` button is pressed, some information from the current logbook is displayed. First the total vertical ascend rate is shown until the `Minus` button is pressed, then the total vertical descend rate and then the number of runs, before the watch returns to the main mode in which it was.

The fast cumulative mode is a typical shortcut and is implemented with an aspect. The pointcut `main-modes-PC` in Figure 8 (a) chooses transitions which have a main mode of the base program as source state and `minus` as input part of the label. Visiting the current logbook is done in several steps: it first displays the ascend rate (output `showAsc`), then the descend rate (`showDesc`), and then the number of runs (`showNbRuns`). Therefore, the aspect outputs first `showAsc` and then inserts the automaton `visit-logbook`, shown in Figure 8(b). As target state, we choose an empty trace from the current state, so that the program continues in the main mode in which it was when the aspect was activated. The aspect for the fast cumulative shortcut is fully specified by `Fast-Cumulative` = (`main-modes-PC`, {`showAsc`}, *toCurrent*, ϵ, `visit-logbook`).

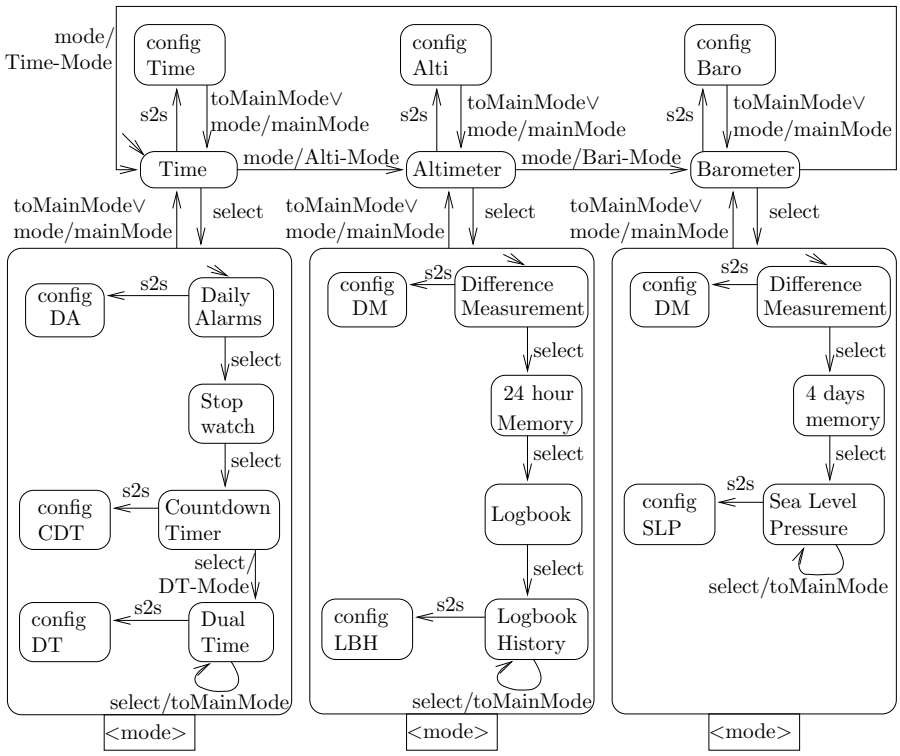


Fig. 7. The `base-program` component. Its interface is: inputs = {`Mode, Select, plus, minus, s2s`}, outputs = {`Time-Mode, Bari-Mode, Alti-Mode, mainMode`}, `toMainMode` is encapsulated.

The Altimax Model. The controller of the Altimax watch is the base program (Figure 7) with the fast cumulative aspect woven to it: `Altimax` = `base-program`◁ `Fast-Cumulative`.

The Compass Mode. We program a controller for the Vector wristwatch by applying three aspects to the base program, which are explained in the sequel. The Vector has a fourth main mode, the compass mode. We add it to the base program with an aspect. The transition going from the Barometer main mode to the Time main mode is the sole join point transition (chosen by the pointcut `baro-mode-PC` in Figure 9 (a)). The only advice output is `Comp-Mode` which displays the compass. The aspect inserts the automaton `Compass` (see Figure 9 (b)), which contains the interface for the compass. After leaving the compass mode, the interface goes back to the `Time` main mode, thus the target state is set to the initial state: this is a *toInit* advice with σ being the empty trace ϵ. The resulting aspect is thus `Compass-Mode` = (`baro-mode-PC`, {`Comp-Mode`}, *toCurrent*, ϵ, `Compass`).

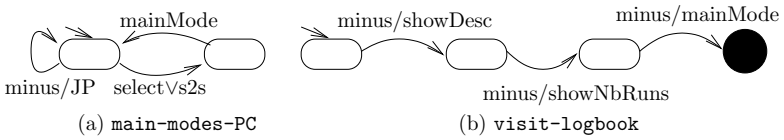


Fig. 8. Pointcut (a) and inserted automaton (b) for the Fast-Cumulative aspect.

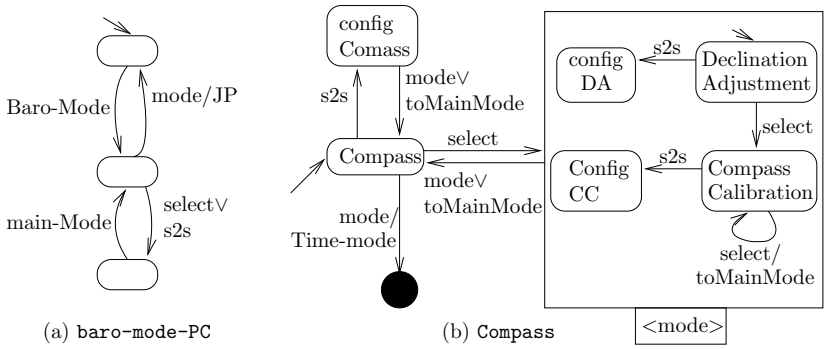


Fig. 9. Pointcut (a) and inserted automaton (b) for the `Compass-mode` aspect

The Compass Shortcut. When the `Minus` button is pressed in a main mode, the Vector does not show information from the current logbook, but goes directly to the compass mode. This is useful when the user is hiking cross-country and wants to check regularly the bearing of the compass. Thus, the Vector does not contain the fast-cumulative aspect, but an aspect that adds advice transitions from the main modes to the compass main mode: `Fast-Compass` = (`main-modes-PC`, {`Comp-Mode`}, *toInit*, `mode.mode.mode.mode`). Note that this aspect must be applied after the `Compass-Mode` aspect, because it uses the compass mode. Indeed, after the compass mode has been added, it can be reached by pressing four times the `Mode` button from the initial state. The trace `mode.mode.mode.mode` ends in state `Compass`; it is the target state of the advice transitions.

No Dual Time Submode. As a last difference with the Altimax, the Vector lacks the Dual Time submode (the fourth submode of the `Time` main mode of the base program in Figure 7), which allows the user to simultaneously view the time in two different time zones. We cut it out of the base program with an aspect. We choose as join points all transitions which emit `DT-Mode`, the signal that tells the underlying component to display the information related to the Dual Time mode. The corresponding pointcut, `Countdown-PC`, consists of a single state with a loop transition with label `DT-Mode/JP`. Instead of going to the Dual Time mode, the Vector goes to the Time main mode, thus the target state is defined by the empty trace. The aspect is thus defined by `No-Dual-Time` = (`Countdown-PC`, {`Time-Mode`}, *toInit*, ϵ).

The Vector Model. The controller for the Vector can thus be built by weaving the three aspects into the base program: `Vector` = `base-program` ◁ `Compass-Mode` ◁ `Fast-Compass` ◁ `No-Dual-Time`.

5 Modular Design with Aspects

Advantages of aspect-oriented programming. We use Argos and Larissa to model the interfaces of two watches. With Larissa, they are modularized in such a way that the common part of the watches (the `base-program`) can be reused, and the behavior that is specific to a single watch can be added with aspects. The interfaces can also be programmed without aspects, but this solution has three principal drawbacks:

1) Programming the shortcuts by hand means to copy/paste the transitions or automata that constitute the shortcut to every main mode.

2) To program the Vector controller based on the Altimax controller, one must, besides adding the Compass Mode and removing the Dual Time Mode, remove the logbook shortcut transitions and states, and add the compass shortcut transitions. This is easy with Larissa (one must just replace the aspect), because the shortcuts are modularized.

3) When programming the Vector without aspects, the automata that contain the main modes and the Time Submode must be copied from the Altimax and modified, leading to code duplication. Thus, if one wants to correct a bug or change something in one of these, the same modification must be applied twice.

Note that the design of aspects itself is modular: the `Fast-Cumulative` aspect has the pointcut program `main-modes-PC` (see Figure 8(a)) which has been reused for the pointcut of the `Fast-Compass` aspect.

Aspects as components. In our setting, we want to consider aspects as normal components. We claim that to be able to do so, the two following properties should hold: *1) aspect weaving should behave as an ordinary composition operator, so as to be freely mixable with ordinary components; 2) the aspect definition should allow component substitutability.* Larissa obeys these properties:

1) The aspect weaving operator ◁ is an ordinary operator of the language. Indeed, weaving an Argos program and an aspect results in another Argos program. This allows the construction of arbitrary expressions made of Argos operators, aspects and programs. For instance, the `Vector` model is obtained by weaving several aspects. This means that the `Compass-Mode` aspect is woven into the `base-program`, producing an Argos program into which the `Fast-Compass` aspect is woven, etc.

2) The aspect definition only refers to the interface of the program it has to be woven into. Thus, we can replace a component by another one with the same interface, and the aspect can still be applied. Moreover, the semantics of the weaving does not depend on the way the program is implemented (e.g. local variables or internal names like state names), but only on its semantics. Thus, when we replace a component with an semantically equal one, we obtain a

semantically equivalent program. For instance, if the `base-program` is replaced by a semantically equivalent, but more efficient one, the `Altimax` and the `Vector` are obtained by replacing the base program by the new one, with the guarantee that they execute as before.

6 Related Work

Concerning automata-based language, the closest work is an extension of Statecharts [12]. Aspects are modeled as normal Statecharts, and a transition in an aspect is taken before or after a certain transition in the base program. This approach does not have the semantical properties we are looking for, but has the advantage of being closer to AspectJ.

As for the integration of aspects and components, interesting approaches have been proposed, e.g. in the ACP4IS workshop. Most approaches, e.g. [4, 18], use aspects as a tool in component-based frameworks, for instance for adapting components to a given context of use. Others, e.g. [15], consider aspects as components which are woven into the components they are assembled with. This is close to our setting, in which aspects are ordinary pieces of programs.

The third direction to which our work relates is the use of AOP to build man-machine interfaces, but we found very few papers there. [17] uses aspect-oriented programming to reduce the constraints imposed by the model-view-controller paradigm, which is central in many man-machine interfaces.

7 Conclusion

With a case-study on the design of small electronic device interfaces, we illustrated the use of the aspect-oriented extension of an automaton-based language for reactive systems. This case-study mainly serves for exploring the idea that aspects should be freely mixed with other kinds of components, and that weaving is a particular assembling mechanism. The first step in this direction is to have a clean and formal semantics of aspects. Aspects that do not refer explicitly to the internals of programs are more likely to be the basis for the definition of *aspect components*.

We think that our automaton-based language can easily be used as the core of a component-based approach for reactive systems, since its programs have well-defined interfaces, and it contains clean notions of encapsulation (information hiding), composition between programs, and substitutability of component behaviors. Furthermore, a number of approaches have been proposed to specify components with *contracts*. A contract is a kind of assume-guarantee predicate that characterizes the behavior of a component. The main point we will study in the near future relates to the specification of contracts, in the idea of [13], for aspect components. This rises many questions, such as: how do we define the behavior of an aspect, independently of the program it is woven with? Can we define a semantic equivalence between aspects, in such a way that aspects are substitutable, as usual components are? We need to clarify those notions before

being able to introduce contracts for aspects and to fully consider aspects as components in our framework.

References

1. K. Altisen, F. Maraninchi, and D. Stauch. Aspect-oriented programming for reactive systems: Larissa, a proposal in the synchronous framework. *Sci. Comput. Programming, Special Issue on Foundations of Aspect-Oriented Programming*, 2006. To appear.
2. Y. Coady and G. Kiczales. Back to the future: A retroactive study of aspect evolution in operating system code. In *AOSD'03*, pages 50–59, 2003.
3. A. Colyer and A. Clement. Large-scale AOSD for middleware. In *AOSD'04*, pages 56–65, 2004.
4. P.-C. David and T. Ledoux. An approach for developing self-adapting fractal components. In *5th International Symposium on Software Composition*, Vienna, Austria, Mar. 2006.
5. N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer Academic Pub., 1993.
6. N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE. *IEEE Trans. Softw. Eng., Special Issue on the Specification and Analysis of Real-Time Systems*, Sept. 1992.
7. N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *3rd Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, June 1993.
8. D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Programming*, 8(3):231–274, June 1987.
9. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *LNCS*, 2072:327–353, 2001.
10. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, SE-3(2):125–143, 1977.
11. Compiler for Larissa. http://www-verimag.imag.fr/∼stauch/ArgosCompiler/.
12. M. Mahoney, A. Bader, T. Elrad, and O. Aldawud. Using aspects to abstract and modularize statecharts. In *5th Aspect-Oriented Modeling Workshop*, 2004.
13. F. Maraninchi and L. Morel. Logical-time contracts for the development of reactive embedded software. In *30th Euromicro Conference, Component-Based Software Engineering Track (ECBSE)*, Rennes, France, Sept. 2004.
14. F. Maraninchi and Y. Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, 27(1/3):61–92, 2001.
15. N. Pessemier, L. Seinturier, T. Coupaye, and L. Duchien. A model for developing component-based and aspect-oriented systems. In *5th International Symposium on Software Composition*, Vienna, Austria, Mar. 2006.
16. D. Stauch, K. Altisen, and F. Maraninchi. Interference of Larissa aspects. In *Workshop on the Foundations of Aspect-Oriented Languages (FOAL)*, 2006.
17. M. Veit and S. Herrmann. Model-view-controller and object teams: A perfect match of paradigms. In M. Akşit, editor, *AOSD'03*, pages 140–149, 2003.
18. E. Wohlstadter, S. Tai, and P. Devanbu. Two party aspect agreement using a COTS solver. In *Proceedings of the Fourth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, Mar. 2005.