Exploring Aspects in the Context of Reactive Systems

Karine Altisen
Verimag/INPG, Grenoble
Centre Equation – 2, avenue de Vignate, F 38610 GIERES
Karine.Altisen@imag.fr

Florence Maraninchi
Verimag/INPG, Grenoble
Centre Equation – 2, avenue de Vignate, F 38610 GIERES
Florence.Maraninchi@imag.fr

David Stauch
Verimag/INPG, Grenoble
Centre Equation – 2, avenue de Vignate, F 38610 GIERES
David.Stauch@imag.fr

ABSTRACT
We explore the semantics of aspect oriented programming languages in the context of embedded reactive systems. For reactive systems, there are a lot of simple and expressive formal models that can be used, based on traces and automata. Moreover, the main construct in the programming languages of the domain is parallel composition, and the notion of *transverse* modification is quite natural. We propose: 1) a semantical definition of an aspect, allowing one to study its impact on the original program; 2) some operational constructs that can constitute the basis of a weaving mechanism.

Categories and Subject Descriptors
D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms
Design, Languages, Theory, Verification

Keywords
reactive systems, aspect-oriented programming, formal semantics, synchronous languages

1. INTRODUCTION

1.1 Generalities
Whatever be the structuring mechanisms offered by a programming language, it is always possible to find a program P and some functionality F that P should provide (or a property P should ensure), in such a way that F cannot be implemented only by some modifications of P that would "respect" its existing structure. F is then called an *aspect* [6]. For instance, adding debugging facilities to a program is an aspect because its implementation is likely to need small modifications everywhere.

FOAL '04 Lancaster, GB

Aspect programming studies how aspects can be *specified*, and how the additional code needed to implement F can be automatically *woven* into P, statically or dynamically. All these definitions are quite informal and, for a given programming language, it is not always clear whether a given functionality is indeed an aspect. This shows that any formal definition of aspects and weaving has to provide precise definitions for, at least: 1) the structure of programs (this is the simplest, but we have to choose between a concrete syntax and a very abstract semantic structure); 2) the notion of a functionality to be implemented in a given program; 3) an aspect specification language; 4) a program transformation (not necessarily static) for the weaving process.

There are very few attempts at defining the notion of aspect formally, independently of any language. It is probably too early, and we need to understand the notion of aspect better, concentrating on specific application domains.

1.2 Objectives of the paper
We would like to question the notion of aspect a bit further, in a formally defined context that allows one to give unambiguous definitions for all the important notions related to aspects. We choose reactive systems because they provide a very clear notion of the input/output interface of a program, their design relies on a parallel composition and a communication mechanism, and the definition of their sequential behavior is simple. Additionally, we will consider *static* weaving mechanisms only, because reactive systems are most of the time *embedded*: a monitoring system that may stop the program when its behavior diverges from the expected one is useless, because the system cannot be repaired on-line. See section 5 for more comments on the relationship with so-called *property-enforcing* techniques that can sometimes be considered as aspect-weaving.

Moreover, we choose a formalism made of synchronous compositions, because the synchronous broadcast and the compiled synchronous parallel composition are already very powerful (for instance, rendez-vous can be implemented by the so-called "instantaneous dialogue"). A a lot of things can be implemented by adding components synchronized with an existing program. Something that cannot be implemented this way really deserves the name of aspect. There exist several synchronous languages with very distinct constructs, that all have a semantics in terms of communicating Mealy machines. Considering aspects at the level of this model guarantees that the definitions are not too particular to a given language.

We try to formalize the following observations, questions and requirements:

(α) What is the semantic impact of weaving? Can the behavior of the modified program be completely distinct from the behavior of the original program? We would like the new program to be somewhat comparable to the old one, at least on a subset of its inputs, and/or on a subset of its instants.

(β) An aspect specification language has to talk about P. How detailed can it be? Can it talk about the interface only, or about the internals of the program?

(γ) How can we be sure that F cannot be implemented in P with simple modifications that do not require the aspect point of view?

The contribution of the paper is a simple formal framework in which aspects of reactive systems can be specified and studied. It is a first approach to this question, and this work can be continued in various ways.

Section 2 defines a formal model for reactive systems; Section 3 is a non exhaustive list of functionalities that constitute good candidates for the notion of aspect in reactive systems; Section 4 is our proposal: a notion of aspects and how to specify them, formal basis for the weaving process; Section 5 is a (probably) non exhaustive list of related work; Section 6 is the conclusion, mainly a list of perspectives.

2. THE MODEL

The formal model for reactive systems is given here in two levels: 1) a trace semantics that is common to a wide variety of languages, and 2) a set of operators on reactive and deterministic Mealy machines that can be considered as an ideal view of a synchronous language. All questions and problems that can be formulated in the first level are therefore general to a lot of languages; the second level will be used whenever we need a *structure* in the programming language.

2.1 Traces and trace semantics

Definition 1 (Traces) *Let $\mathcal{I}$, $\mathcal{O}$ be sets of Boolean input and output variables representing signals from and to the environment. A* trace on $\mathcal{I} \cup \mathcal{O}$, *$t$, is a function: $t : \mathbb{N} \longrightarrow [(\mathcal{I} \cup \mathcal{O}) \longrightarrow \{\mathtt{true}, \mathtt{false}\}]$. At each instant $n \in \mathbb{N}$, the given trace t provides the valuations of every input and output.*

A set of traces $st = \{t \mid t$ is a trace on $\mathcal{I} \cup \mathcal{O}\}$ is deterministic *iff*

$$\forall t, t' \in st. \big(\forall n \in \mathbb{N}. \forall i \in \mathcal{I}. t(n)[i] = t'(n)[i]\big) \implies \big(\forall o \in \mathcal{O}. t(n)[o] = t'(n)[o]\big).$$

We write $\{i_1, i_2, ..., i_{\|\mathcal{I}\|}\}$ for the set of inputs $\mathcal{I}$. A set of traces $st = \{t \mid t$ is a trace on $\mathcal{I} \cup \mathcal{O}\}$ is reactive *iff*

$$(i)\ \forall (v_1, v_2, ..., v_{\|\mathcal{I}\|}) \in \{\mathtt{true}, \mathtt{false}\}^{\|\mathcal{I}\|}. \exists t \in st. \forall k. t(0)[i_k] = v_k$$

$$(ii)\ \forall t \in st. \forall n \in \mathbb{N}. \forall (v_1, v_2, ..., v_{\|\mathcal{I}\|}) \in \{\mathtt{true}, \mathtt{false}\}^{\|\mathcal{I}\|}. \exists t' \in st. (\forall m \leq n. \forall i \in \mathcal{I}. t(m)[i] = t'(m)[i]) \wedge (\forall k. t'(n+1)[i_k] = v_k)$$

A set of traces is a way to define the semantics of a program P, given its inputs and outputs. From the above definitions, a program P is *deterministic* if from the same sequence of inputs it always computes the same sequence of outputs. It is *reactive* whenever it allows every sequences of every eligible valuations of inputs to be computed. Determinism is related to the fact that the program is indeed written with a programming language (which has deterministic execution); reactivity is an intrinsic property of the program that has to react forever, to every possible inputs without any blocking.

Definitions 2 and 3 below define transformations on traces that will be used to characterize the impact of weaving on the semantics of a program.

Definition 2 (Masking traces) *Given a trace t on $\mathcal{I} \cup \mathcal{O}$ and a subset $S \subseteq (\mathcal{I} \cup \mathcal{O})$ the* masking *of t by $(\mathcal{I} \cup \mathcal{O}) \setminus S$ is a trace $t_{|S}$ on S such that:*

$$\forall e \in S. \forall n \in \mathbb{N}. t_{|S}(n)[e] = t(n)[e].$$

Definition 3 (Clocking traces) *Given a trace t on $\mathcal{I} \cup \mathcal{O}$ and a subset $M \in \mathbb{N}$ the* clocking *of t by M is a trace $t_{\|M}$ on $\mathcal{I} \cup \mathcal{O}$ such that:*

$$\forall e \in (\mathcal{I} \cup \mathcal{O}). \forall n \in \mathbb{N} \setminus M. t_{\|M}(n)[e] = t(n)[e].$$

Notice that given a trace t and a subset M of natural, this defines a set of clocking including t itself.

Those definitions are naturally extended to sets of traces and programs.

2.2 Elements of language definition

The core of a synchronous language is made of input/output automata, the synchronous product, and the encapsulation operation.

Definition 4 (Automaton) *An* automaton *$\mathcal{A}$ is a tuple $\mathcal{A} = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ where $\mathcal{Q}$ is the set of states, $s_{init} \in \mathcal{Q}$ is the initial state, $\mathcal{I}$ and $\mathcal{O}$ are the sets of Boolean input and output variables respectively, $\mathcal{T} \subseteq \mathcal{Q} \times \mathcal{B}ool(\mathcal{I}) \times 2^{\mathcal{O}} \times \mathcal{Q}$ is the set of transitions. $\mathcal{B}ool(\mathcal{I})$ denotes the set of Boolean formulas with variables in $\mathcal{I}$. For $t = (s, \ell, O, s') \in \mathcal{T}$, $s, s' \in \mathcal{Q}$ are the source and target states, $\ell \in \mathcal{B}ool(\mathcal{I})$ is the triggering condition of the transition, and $O \subseteq \mathcal{O}$ is the set of outputs emitted whenever the transition is triggered. Without loss of generality, we consider that automata only have complete monomials as input part of the transition labels.*

The semantics of the automaton $\mathcal{A} = (\mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ is given in terms of a set of traces on $\mathcal{I} \cup \mathcal{O}$. A trace t of $\mathcal{A}$ is inductively built as follows:

- at time 0, the automaton is in state s_{init};
- if at time $n \in \mathbb{N}$, the automaton is in state $s \in \mathcal{Q}$ and receives input valuations v_i for $i \in \mathcal{I}$, then at time $n+1$ it reaches state s', emitting O:

$$\big(\forall i \in \mathcal{I}.t(n)[i] = v_i \in \{\texttt{true}, \texttt{false}\} \wedge \\ \forall o \in \mathcal{O}.t(n)[o] = w_o \in \{\texttt{true}, \texttt{false}\}\big) \implies \\ \big(\exists (s, \ell, O, s') \in \mathcal{T} \wedge O = \{o \in \mathcal{O}, w_o = \texttt{true}\} \wedge \\ \ell \text{ valuates to true w.r.t. the } v_i\text{'s}\big).$$

We note *Trace*($\mathcal{A}$) the set of all traces built following this scheme: *Trace*($\mathcal{A}$) defines the semantics of $\mathcal{A}$. The automaton $\mathcal{A}$ is said to be *deterministic* (resp. *reactive*) iff its set of traces *Trace*($\mathcal{A}$) is deterministic (resp. reactive) (see def. 1).

Definition 5 (Synchronous Product) *Let $\mathcal{A}_1 = (\mathcal{Q}_1, s_{init1}, \mathcal{I}_1, \mathcal{O}_1, \mathcal{T}_1)$ and $\mathcal{A}_2 = (\mathcal{Q}_2, s_{init2}, \mathcal{I}_2, \mathcal{O}_2, \mathcal{T}_2)$ be automata. The* synchronous product *of $\mathcal{A}_1$ and $\mathcal{A}_2$ is the automaton $\mathcal{A}_1 || \mathcal{A}_2 = (\mathcal{Q}_1 \times \mathcal{Q}_2, (s_{init1} s_{init2}), \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{T})$ where $\mathcal{T}$ is defined by:*

$$(s_1, \ell_1, O_1, s_1') \in \mathcal{T}_1 \wedge (s_2, \ell_2, O_2, s_2') \in \mathcal{T}_2 \Longleftrightarrow \\ (s_1 s_2, \ell_1 \wedge \ell_2, O_1 \cup O_2, s_1' s'2) \in \mathcal{T}.$$

The synchronous product of automata is both commutative and associative, and it is easy to show that it preserves both determinism and reactivity.

Definition 6 (Encapsulation) *Let $\mathcal{A} = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ be an automaton and $\Gamma \subseteq \mathcal{I} \cup \mathcal{O}$ be a set of inputs and outputs of $\mathcal{A}$. The* encapsulation *of $\mathcal{A}$ w.r.t. Γ is the automaton $\mathcal{A} \setminus \Gamma = (\mathcal{Q}, s_{init}, \mathcal{I} \setminus \Gamma, \mathcal{O} \setminus \Gamma, \mathcal{T}')$ where $\mathcal{T}'$ is defined by:*

$$(s, \ell, O, s') \in \mathcal{T} \wedge \ell^+ \cap \Gamma \subseteq O \wedge \ell^- \cap \Gamma \cap O = \emptyset \Longleftrightarrow \\ (s, \exists \Gamma.\ell, O \setminus \Gamma, s') \in \mathcal{T}'$$

ℓ^+ is the set of variables that appear as positive elements in the monomial ℓ (i.e. $\ell^+ = \{x \in \mathcal{I} \mid (x \wedge \ell) = \ell\}$). ℓ^- is the set of variables that appear as negative elements in the monomial l (i.e. $\ell^- = \{x \in \mathcal{I} \mid (\neg x \wedge \ell) = \ell\}$).

Intuitively, a transition $(s, \ell, O, s') \in \mathcal{T}$ is still present in the result of the encapsulation operation if its label satisfies a local criterion made of two parts: $\ell^+ \cap \Gamma \subseteq O$ means that a local variable which needs to be true has to be emitted by the same transition; $\ell^- \cap \Gamma \cap O = \emptyset$ means that a local variable that needs to be false should *not* be emitted in the transition.

If the label of a transition satisfies this criterion, then the names of the encapsulated variables are hidden, both in the input part and in the output part. This is expressed by $\exists \Gamma.\ell$ for the input part, and by $O \setminus \Gamma$ for the output part.

In general, the encapsulation operation does not preserve determinism nor reactivity. This is related to the so-called "causality" problem intrinsic to synchronous languages (see, for instance [2]).

An example

Figure 1-(i) shows a 3-bits counter. Dashed lines denote parallel compositions and the overall box denotes the encapsulation of the three parallel components, hiding signals `b` and `c`. The idea is the following: the first component on the right receives `a` from the environment, and sends `b` to the second one, every two `a`'s. Similarly, the second one sends `c` to the third one, every two `b`'s. `b` and `c` are the carry signals. The global system has `a` as input and `d` as output; it counts `a`'s modulo 8, and emits `d` every 8 `a`'s. Applying the semantics of the operator (first the product of the three automata, then the encapsulation) yields the simple flat automaton with 8 states (Figure 1-(ii)).

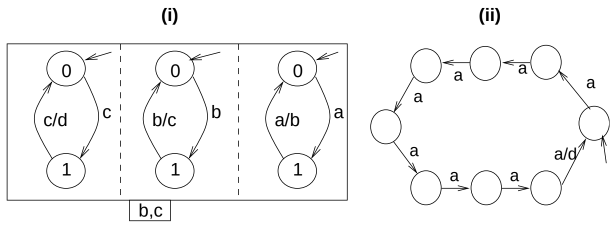

Figure 1: A 3-bits counter. Notations: in each automaton, the initial state is denoted with a little arrow; the label on transitions are expressed by "`triggering cond. / outputs emitted`", e.g. the transition labelled by "`a/b`" is triggered when `a` is true and emits `b`.

3. EXAMPLES OF ASPECTS FOR REACTIVE SYSTEMS

Among the traditional programming examples in the domain of reactive systems, there are some typical cases in which one could think of introducing aspect programming. The candidate "aspects" for reactive systems sometimes depend on the programming paradigm of the language used, but not always. We give several examples below.

3.1 Conditional reinitialization

Being able to re-initialize a reactive system on the occurrence of a special signal is useful in several contexts. For instance, if we program two different systems S1 and S2, in isolation, and then want to design a more complex one in which the two of them have to be used, it is likely that one of the systems is active in a given period of time, then the other one, then the first one again. When the first system restarts, it should have the same behavior as if it were started for the first time. Making a reactive program reinitialiazable means adding reactions to a special signal r, that leads the system to its initial configuration, from any other state it may have reached.

In some languages, there is a dedicated construct for this. In Esterel [2], if `S` is the program of the original system, and `r` the additional signal, an expression similar to "`loop S each r`" is the reinitializable program. This is true for a main program, and this is also true for a subprogram in a complex context: `S` can be replaced by `loop S each r` without changing the context.

In Lustre [1], which is not based upon the imperative paradigm, reiniatializing a program needs modifications everywhere in the code. Indeed, there is a special conditional structure whose meaning is: *if (this is the beginning of time) then ... else ....* When a program has to be made reinitializable by a signal `r`, all occurrences of this special construct have to be modified to give: *if ((this is the beginning of time) OR r is present) then ... else ....*

In the language of parallel automata we presented in section 2.2, making an automaton reinitializable means adding transitions from any state to the initial one, with the condition `r`, and no emitted signal; moreover, all the existing transitions have their condition reinforced by `not r`. Notice that, in a more sophisticated language based on explicit automata, one would introduce hierarchy of states à-la Statecharts, and reinitialization would become trivial (see, for instance [7]).

3.2 Conditional inhibition

Conditional inhibition means the following: when an additional signal `ck` is present, the new system behaves as the old one. When this signal is not present, then the system does not evolve, it keeps its current state, until `ck` is present again.

In dataflow languages for reactive systems like Lustre or Signal, this kind of behavior is a special case of a system with multiple *clocks*. There are dedicated constructs that allow to manipulate clocks. In Lustre, if `S` is the original program, then something like "`current (S when ck)`" would do the job.

In our language of automata, adding conditional inhibition means (1) enforcing the triggering condition of each transition by $\wedge\neg ck$ and (2) adding to each state s a self-loop transition $(s, ck, \emptyset, s)$ with triggering condition ck and emitting nothing.

3.3 Adding a validity bit

This last example is very common in fault-tolerant systems. Consider a reactive system with an input `i` and an output `o`. `i` comes from an external physical device, that may "fail" temporarily. Some physical properties make it possible to produce the information: *the value transmitted on input* `i`*, in the current instant, is not valid.*

The problem is to rewrite the program, adding a *validity bit* `v` to the input `i`. The new program should behave as the original one when its input is valid (i.e. when `v` is true). When `v` is false, the value of `i` should not be "*taken into account*". This quite informal specification rises two questions.

First, forbidding the use of the input at a given instant in time means that: a) we have to give a default value for the outputs that depend on it; b) we also have to make sure that the input does not serve for updating the memory.

Second, in all cases, one has to determine whether the value computed for the outputs, or the way memory is updated, really *depends* on the input `i` in the current instant. If it is possible to write an additional parallel component that observes `i` and each output `o`, and delivers a Boolean signal `o_depends_on_i`, then the program can be modified quite easily, with transformations similar to the ones we described for reinitialization or inhibition (this is similar for memory updates). But writing this additional component is difficult: `o_depends_on_i` means *if we change* `i`*, then* `o` *changes*, and it cannot be computed by observing *a single* input/output sequence, but the whole set of traces. So it cannot be implemented by an additional parallel component.

All transformations that mention such a dependency notion between inputs and outputs are good examples for point γ in the introduction: we do not know how to program them with the usual constructs of our languages, and it should even be possible to prove that we cannot do so. See more comments in section 6.

4. OUR PROPOSAL

Our proposal is made of two parts: first, we characterize the notion of aspect in a very declarative setting. Second, we propose a set of elementary transformations on automata that may be the basis of a weaving mechanism.

4.1 Characterizing the semantical influence of aspects

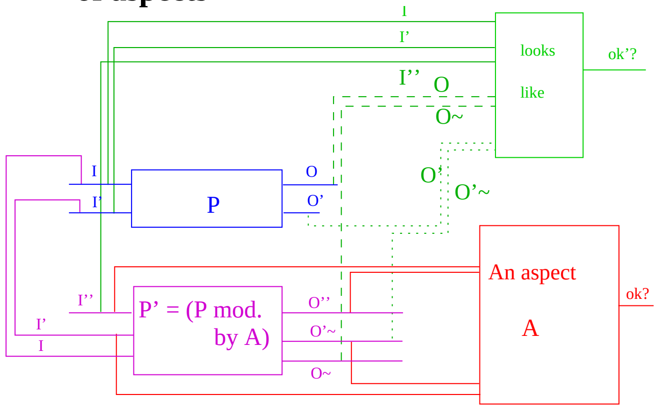

Figure 2: The semantic scheme is a data flow block diagram using the same notations as in the text; the inputs of a box enter the box at its left and the outputs goes out at its right; the box for the aspect *Asp* and the one for the predicate *looks_like* are given in terms of observers, i.e. they only have an output *ok* (resp. *ok'*) that is expected to be always true. $\tilde{O}$ (resp. $\tilde{O'}$) denotes a syntactic copy of the set of outputs O (resp. O'): P emits $O \cup O'$ while P'emits $\tilde{O} \cup \tilde{O'}$ which may not have the same values.

Figure 2 summarizes the semantic framework in which we define aspects: P is a program (a set of traces on its inputs and outputs). *Asp* is an aspect, allowed to deal with some inputs $I' \subseteq \mathcal{I}$ and some outputs $O' \subseteq \mathcal{O}$ of P; it is also allowed to add some inputs I'' and outputs O'' to P. *Asp* is given as a set of traces on $I' \cup I'' \cup O' \cup O''$. Weaving *Asp* into P yields a modified program $P' = P \triangleleft Asp$ such that:

- P' has $\mathcal{I} \cup I''$ as inputs and $\mathcal{O} \cup O''$ as outputs;

- P' is *consistent* with *Asp*, i.e.,

  $$P'_{|I' \cup I'' \cup O' \cup O''} \subseteq Asp;$$

  the set $I' \cup I'' \cup O' \cup O''$ on which P' and Asp are compared is exactly the set of inputs/outputs of Asp; this means that P' on this set must be some of the traces defined by Asp.

The two items above define P' w.r.t. Asp but they do not at all constrain P' to be related to P, in any sense: for example, take for P' every traces in Asp, and extend them by adding the inputs I and outputs O with arbitrary values (e.g. `false` everywhere); it yields P' such that $P'_{|I' \cup I'' \cup O' \cup O''} \subseteq Asp$. This is caricatural, but it illustrates a major semantic problem of aspects from our point of view: we implicitly want P' to be related to P. This is the motivation for our third point.

Third, we require that P and P' be comparable. *looks_like* (P, P') is a predicate that compares the traces of P and P' for the inputs and outputs they have in common: it compares $P_{|\mathcal{I} \cup \mathcal{O}}$ with $P'_{|\mathcal{I} \cup \mathcal{O}}$. Different classes of programs and aspects may require different definitions for *looks_like* . Here are some examples:

- We may require that P and P' have the same traces on the inputs/outputs that are not involved in the definition of the aspect: *looks_like* $\overset{\text{def}}{=} P_{|I \cup O} = P'_{|I \cup O}$;
- one might enforce this condition by imposing that P and P' have the same behavior on $I' \cup O'$ at the instants when Asp does not modify them. Suppose, for example that Asp modifies the value of an output $o \in O'$ if a new input $i \in I''$ is `false`, as in the validity bit example. The definition of *looks_like* may then impose that o keeps its value whenever i is `true`; this condition is expressed by comparing the clocking of the traces of P' on the set of instants when i is `true` with the same clocking applied to P;
- another admissible comparison criterion is that the traces (masked or clocked) of P' are shifted by $n \in \mathbb{N}$ instants compared with those of P. This might be useful in case the aspect itself introduces this kind of shifting.

This list of comparison criteria is not exhaustive and a mix of the three solutions mentioned above as well as different ways of comparing traces of P and P' may be admissible.

4.2 Operational definition of aspects

This part describes an elementary transformation on automata proposed as a basic construct for a weaving process. It was motivated by some of the aspects one may imagine for reactive systems.

4.2.1 A Stateless Transformation

Consider an automaton $\mathcal{A} = (\mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O}, \mathcal{T})$. The idea is to modify the transitions sourced in a given set of states, by reinforcing their condition and adding some emitted events. Whenever a transition condition is reinforced, it means that the disjunction of conditions, for all transitions sourced in the same state, may no longer be "true" (the automaton is no longer reactive). To ensure that the result of the transformation is indeed reactive, we also add a transition with the missing condition. It could be a loop on the state, but it seems more general to allow any existing state to be its target state. The result is not always deterministic.

The parameters of the transformation are: a specification ψ of a set of traces on $\mathcal{I} \cup \mathcal{O}$; a condition C on $\mathcal{I}$; a condition m' on additional inputs; a set o' of additional outputs; a set o'' of additional outputs; a specification ψ' of a set of traces on $\mathcal{I} \cup \mathcal{O}$.

For any state q reachable from the initial state by a trace in ψ, for any of its outgoing transitions $q \xrightarrow{m/o} q'$ whose condition m satisfies C, we transform it into $q \xrightarrow{m \wedge m'/o \cup o'} q'$, and we add $q \xrightarrow{m \wedge \neg m'/o''} q''$ for all q'' reachable from the initial state by a trace in ψ'.

4.2.2 Implementing the Examples

Reinitialization of a single automaton by r is obtained by applying this mechanism with the following parameters: $\psi =$ `true` (any sequence of inputs and outputs is accepted by ψ, meaning: all the reachable states); $C =$ `true` (for all transitions); $m' = \neg r$ and $o' = \emptyset$ (reinforcing the existing transitions); $m'' = r$ and $o'' = \emptyset$ (adding the "reset" transitions); $\psi' = \epsilon$ (specifies the initial state).

It is easy to show that any program made of automata, parallel compositions and encapsulations, can be made reinitializable by r. It is sufficient to apply the above transformation on each of the automata. r should be a fresh variable name, not used in the original program.

The conditional inhibition can be implemented with a similar transformation. Adding a validity bit is more complex, because it involves the notion of "*the transition really depends on the input at the current instant*" and this cannot be expressed by reinforcing conditions on existing transitions.

4.3 The Global Picture

Putting together the informal examples of section 3, the declarative point of view of section 4.1, and the operational view of section 4.2, we obtain the following:

(A) A set of aspect candidates: i.e. program transformations taken from classical examples of reactive programming. On this set of aspects, we may wonder whether we really need something new to implement them: do they deserve the name of aspect, or were there already implementable with traditional program constructs?

(B) A declarative characterization of the relationships between a program P, an aspect A, a new program P' resulting from the weaving of P and A, and a *comparison* criterion used to "control" how much P' can differ from P. This semantic scheme may be used to define a set of P' from P, A and the comparison criterion, but not in an operational way; moreover the set of P' that fit in the picture may be empty.

Finally the comparison criterion should not be completely ad-hoc for an aspect A. There could be several generic comparison criteria, depending on the type of applications. For instance, some applications in reactive programming may allow a modified program to be slightly shifted from the original one (giving the same outputs, but later), while others may not.

(C) A set of elementary program transformations (given on explicit automata) that could constitute the basis of a weaving mechanism.

This setting has some internal consistency requirements. For instance, each elementary transformation proposed in (C) should be such that the transformed program and the original one are comparable in the sense defined by the "looks-like" box in (B).

Once these internal consistency requirements have been expressed and verified on the general setting, the following questions make sense:

– From an informal notion of aspect taken from (A), how to derive the necessary transformations in (C)? We did this for the examples, in a very informal way.

– How to use the framework of (B) to specify one of the candidate aspects of (A)?

– From a formal specification of an aspect in the style of (B), how to produce a sequence of (C) transformations to be applied to P, in order to produce a program P' that fits in the (B) picture?

Finally, some of the general questions people ask about aspects can be expressed in the same semantical framework:

– If we take two trace-equivalent programs P_1 and P_2 and an aspect A, is it true that $P_1 \lhd A$ and $P_2 \lhd A$ are still trace-equivalent?

– How to compare $P \lhd A_1 \lhd A_2$ and $P \lhd A_2 \lhd A_1$? This is related to the more general notion of aspect interference.

5. RELATED WORK

Related work can be found in several directions. First, papers from the AOP community, with emphasis on semantics; the setting of [9] is very close to ours, since it considers sets of parallel input/output automata; however, synchronization is made by shared variables, and the author considers transitions made of sequences of elementary actions, on which the aspectJ-like "insert-before" or "insert-after" transformations make sense. In the above paper, the notions of "property inheritance" and "property superimposition" are defined. They seem to be particular cases of the declarative scheme of section 4.1, relating properties of $P \lhd A$ with properties of P (inheritance"), or A (superimposition"). Superimposition itself was introduced earlier (see [4]), ranging from *spectative* techniques (mere *observers*) to *invasive* techniques. Our work could also be called: static invasive superimposition.

Second, papers on controller synthesis techniques [10] and their application to interfaces [3] in component-based designs, because the definition of P' from P, the aspect specification and the "looks-like" predicate (shown in section 4.1) can be viewed as a controller-synthesis problem.

Third, there have been a number of papers on "property-enforcing techniques" that can be expressed in a aspect-oriented style. The idea is to express safety properties (e.g. with explicit automata like in [8]) and to "run" them in parallel with a program. When the safety property becomes false, the program is stopped. Running them in parallel means performing a product between the program and the property, and can be viewed as dynamic weaving. In our setting, performing the product between the program and a safety property does not need the notion of aspect: the safety property can be expressed as a subprogram in parallel with the original program P, and the whole can be compiled. This yields a new program Q with an output ok meaning: the property is true from the beginning of time. However, for embedded systems, waiting execution-time to know about a safety property is not a solution. Static verification techniques are used instead.

6. CONCLUSION

The semantic setting we presented is adequate for the questions α and β of the introduction, and their answers. We are currently investigating the various notions presented here on some simple examples similar to the reinitialization. The idea is not to obtain an automatic method from the semantic picture of figure 2: it is an instance of controller-synthesis, known to be quite costly. Moreover, we presented Boolean programs only, but the general interesting case uses integer variables in reactive programs. The problem then becomes undecidable. The semantic scheme is there only to give a clear meaning to the operational mechanisms we may invent in order to obtain automatic transformation techniques. The examples will probably lead to other basic transformations on automata. A natural extension of the one we presented would allow the creation of new states.

Question γ of the introduction has already been discussed about the validity bit example. We need to define precisely the notion of *dependency* between inputs and outputs in a reactive program, and to prove that it cannot be programmed with the usual constructs, thus deserving the name of aspect. Similar *dependency* notions (like the one of [5], developed for studying security policies) seem worth investigating.

7. REFERENCES

[1] J.-L. Bergerand, P. Caspi, N. Halbwachs, D. Pilaud, and E. Pilaud. Outline of a real time data-flow language. In *Real Time Systems Symposium*, San Diego, Sept. 1985.

[2] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[3] L. de Alfaro and T. A. Henzinger. Interface automata. In V. Gruhn, editor, *Proceedings of the Joint 8th European Software Engeneering Conference and 9th ACM SIGSOFT Symposium on the Foundation of*

Software Engeneering (ESEC/FSE-01), volume 26, 5 of *SOFTWARE ENGINEERING NOTES*, pages 109–120, New York, Sept. 10–14 2001. ACM Press.

[4] N. Francez and I. R. Forman. Superimposition for interacting processes. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR '90: Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 230–245, Amsterdam, The Netherlands, 27–30Aug. 1990. Springer-Verlag.

[5] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 11–20, Oakland, CA, Apr. 1982. IEEE Computer Society Press.

[6] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[7] F. Maraninchi and Y. Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, (27):61–92, 2001.

[8] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, Feb. 2000.

[9] H. B. Sipma. A formal model for cross-cutting modular transition systems. In *Proceedings of the workshop on Foundations of Aspect-Oriented Languages*, Northeastern University, Boston, Mar. 2003.

[10] W. M. Wonham and P. J. Ramadge. On the supremal controllable sublanguage of a given language. *SIAM Journal on Control and Optimization*, 25(3):637–659, May 1987.