
Observation-based interaction and concurrent aspect-oriented programming

Iulian Ober and Younes Lakhrissi

Université de Toulouse - IRIT
118 Route de Narbonne, 31062 Toulouse, France
E-mail: iulian.ober@irit.fr

Summary. In this paper we propose the use of *event observation* as first class concept for the composition of software components. The approach can be applied to any language based on concurrent components, and we illustrate it with examples on a concrete language (Omega UML) used in the specification of real-time systems. To motivate the proposal, we discuss how it may be used to support a very general form of aspect-oriented programming.

1 Introduction

In this paper we study the use of *event observation* as interaction mechanism between software components. The main part of the paper is concerned with the general concepts that are behind observation-based interaction. However, in the final part we are also concerned with motivating why and how this form of interaction is useful. Although other applications are possible, we particularly concentrate on how it may be used to support aspect-oriented programming.

Traditionally, event observation is employed in system modelling in very specific contexts, for example for expressing *properties* that are to be satisfied by a model or a software artifact. The origin of observers as a property specification formalism can be traced back to the Veda tool [13], and the concept has over the years proved to be both powerful and intuitive to use by non-experts in formal verification and has encountered a certain success in several other verification tools, both industrial [1] and academic [3, 4]. Our previous work [9] is a first attempt to use event observation as basis for the specification of certain aspects of a (real-time) software system, more precisely the aspects pertaining to the timing of the system execution.

In the present paper, we generalize our model from [9] in order to use event observation for behavior specification and composition (in [9] observation is only used for specifying duration constraints between events). Additionally, in this paper we propose a systematic approach for deriving event types and the definition of observations from the operational semantics of the host language to which they are added, whereas in [9] observations and event types are defined more or less ad hoc. We analyze in this paper the following questions:

- How can *event observation* be smoothly integrated in a traditional computation model (the *host* model) based on communicating objects? This involves several subsidiary questions:

What is the set of events (\mathcal{E}) that may be generated by the components of a system? There is no general answer to this since it depends on the specifics of the *host* model, but in §2.1 we suggest where to start searching.

How to define the language constructs allowing a component to observe (sets of) events generated by other components? Such a construct has to capture information related to *event types* (e.g., distinguish an object creation event from a method call event), to the *scope of observation* (e.g., local to a component, global), to the existence of specific *event data* (e.g., method parameters for a method call event). The *observation* construct we define in §2.2 tries to answer these questions.

How to compose behaviors based on observations? We tackle this question in §3.

- Once a computation model integrating observation-based interaction is defined, how can it be put to use in practice? In §4 we show that such a model can support aspect-oriented software development in a very general way.

The concrete host language that we consider in our study is a particular profile of UML [18], Omega UML [6, 9]. This choice is justified by the fact that, unlike the standard UML language, Omega UML has a well defined operational semantics given in [6] and in particular a very clear concurrency model based on a notion of active objects. Also motivating us was the fact that we previously developed execution and verification tools for the profile, which give us the possibility to rapidly prototype the concepts proposed in this paper. Nevertheless, we are concerned with the generality of our proposal and we strongly believe that the ideas presented here can be adapted to other host languages based on different computation models. Such issues are left for future exploration.

The rest of the paper begins with a short presentation of the Omega UML computation model. Afterwards, the structure of the paper is guided by the questions given previously in this section. We end it by comparing our approach to some existing proposals, and by drawing conclusions and the main lines of future work.

1.1 The Omega UML profile

The Omega profile [6, 9] can be described as a *semantic profile* of UML, in the sense that it defines very few extensions (i.e., new language constructs), and it mainly concentrates on making precise the semantics of the existing concepts of UML. The focus is specifically on the concepts and diagrams which serve to build operational (executable) design models: class models and their associated behavior (operations and state machines). The other elements of UML (use cases, interactions, activities, deployment, etc.) are not forbidden in Omega UML models, but they are not particularly specialized, nor given a more precise semantics.

The profile specializes the semantics of UML mainly in two respects:

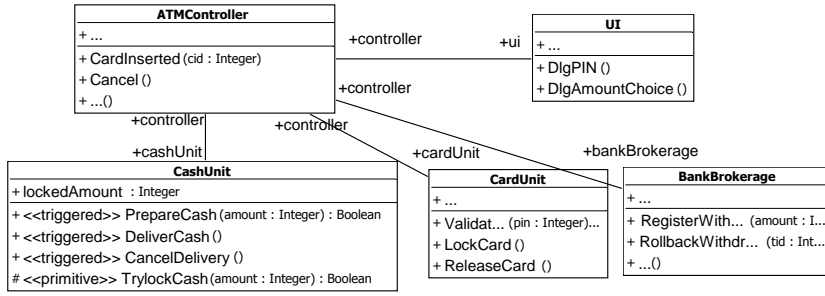


Fig. 1. ATM example: class diagram

- The *computation model*, i.e. the runtime semantics of notions like active and passive objects (including their relationship to control threads), operation invocation, signal exchange, timing constructs, and others. Note that in this respect, the standard definition of UML intentionally leaves many questions open – these are called semantic variation points.
- The *action language* used for fine-grained description of object behavior in operations or on state machine transitions. Note that UML defines an *action semantics* with all the necessary action types, but without any concrete syntax. Therefore, any concrete application (e.g., design tool) using this part of the language has to define its own concrete language.

The choices made in Omega UML with respect to these two points are briefly outlined in this section. We emphasize that the justification of the choices made in the profile (which derive from its application domain – the design of verifiable real-time systems), is outside the scope of this paper and may be found in [6, 9]. We describe the profile only to the extent necessary to understand the examples given in the paper.

The Omega UML computation model

The OMEGA computation model is an extension of the computation model of the Rhapsody UML tool (see [10] for details). It is based on the existence of two kinds of classes: *active* and *passive* ones. At runtime, each instance of an active class defines a concurrency unit with one thread of control, called an *activity group*. Each passive class instance belongs to exactly one activity group, the one of the instance that created it.

Apart from defining the partition of the system into activity groups, there is no difference between how active and passive classes (and instances) are defined and handled. Both kinds of classes are defined by their *attributes*, *relationships*, *operations* and *state machine*, and their operational semantics is identical.

Different activity groups are considered as *concurrent*, and each activity group treats external requests (signals and operation calls from outside the group) one by one in a run-to-completion fashion, in order to protect itself from interference of concurrent requests. This protection policy is common in many concurrent computation models, including Hoare’s monitors, Ada protected objects, SDL

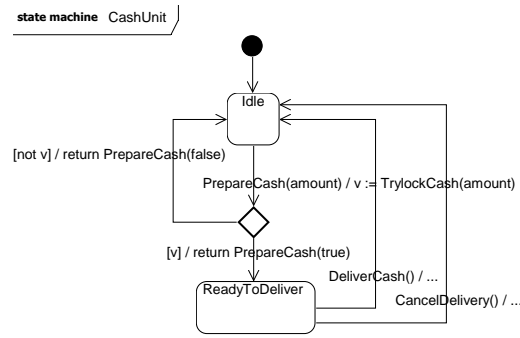


Fig. 2. ATM example: state diagram of `CashUnit`

processes, etc., the particularity of Omega UML being that it is applied to a group of objects instead of only one.

As object interaction mechanism, Omega UML supports only *synchronous* operation calls, where the caller object and its activity group are blocked in a *suspended* state until the explicit return of control from the callee. Two kinds of operations are defined:

- *Triggered operations* which are not allowed to have a body and for which the behavior is described directly in the state machine of the class (the operation call is seen as a special kind of transition trigger).
- *Primitive operations* which are described by a body, like usual methods in object oriented programming languages.

The Omega action language

In order to describe a meaningful behaviour for a UML model, one also needs to describe *actions*: the effect of a transition from a state machine or the body of an operation. The OMEGA profile defines a textual action language compatible with UML action semantics [18], which covers *object creation and destruction*, *operation calls*, *expression evaluation*, *assignments*, *return* from operations, as well as control flow structuring statements (*conditionals* and *loops*). The concrete syntax is self explanatory as it relies on conventions widely used in imperative object oriented languages (Ada, C++, etc.).

An example

To illustrate the previous description we consider a small yet relatively complete example of an ATM machine in Omega UML. The structure of the model, composed of five classes, is shown in Fig. 1. The model is such that each class has exactly one instance at run-time; moreover, although it is not apparent in the figure, all classes are *active* – each of their instances will have its own thread of control. The structure of the classes is as usual given by attributes and relationships (associations). The behavior of each class is specified by its state machine and possibly by its *primitive* operations.

For brevity, in this example we focus in particular on one class, the `CashUnit`. Its interface is visible in Fig. 1: it contains three public triggered operations called by the `ATMController` (`PrepareCash`, `DeliverCash`, `CancelDelivery`), and one private primitive operation `TrylockCash` which groups some internal actions executed by this object (checking the cash store for availability of a requested `amount` and if ok, locking the `amount` for delivery).

The behavior of the `CashUnit` object is partially visible in Fig. 2. Initially it is in state `Idle`. Upon reception of a triggered operation call (`PrepareCash`) it calls the internal operation `TrylockCash`. Depending on the success of this operation, it may either return `false` to the caller of `PrepareCash` and go back to `Idle` state, or return `true` go to state `ReadyToDeliver`. In `ReadyToDeliver`, the object may either receive a `DeliverCash` call which causes the delivery of the locked cash (details are omitted), or it may receive a `CancelDelivery` call which causes it to restore the locked cash (this can be decided by the `ATMController` for various reasons).

2 Events and the observation construct

In this section we define the two essential notions of our proposal:

- *event* – a run-time entity which designates the occurrence of a particular condition in the execution of a software system/component, together with the relevant data for that occurrence.
- *observation* – a language construct which, for a given system execution, corresponds to an ordered set of *events* and allows to refer to these events in the system model (e.g., as triggers for some behavior, etc.)

2.1 Events

We relate *events* to the smallest (i.e. indivisible) state changes described by the semantics of a model/language. We consider in particular the case of languages provided with a *structural operational semantics* (SOS, in the sense of [19]), where events therefore correspond to the transitions of the semantic *labeled transition system* (LTS) associated with a model.

SOS terminology

In SOS the semantics of a program (or system model) is a labeled graph (LTS) whose vertices represent “*global states*” of the program, and whose edges (also called *transitions*) represent the smallest (atomic) steps executed by the program to go from one state to another. The graph paths which start in an identified *initial* global state represent the possible *executions* of the program.

For example, for a concurrent object-oriented language like Omega UML, a global state includes the attribute values, state and request queue for all existing objects, as well as execution context (call stack, etc.) for all threads (activity groups). An LTS transition corresponds to the execution of an individual action

by one object, such as: consuming an operation request from the queue, starting to fire a state machine transition, executing an assignment, issuing an operation request, etc. Note that an LTS transition is not to be confused with a state machine transition (from the state machine associated to a UML class): the latter is usually executed as a sequence of LTS transitions.

The LTS corresponding to a program is usually not defined explicitly but rather implicitly, by a series of rules that may be used to construct it inductively. A *global state* is represented by an algebra whose signature obeys to some (naming) rules which give it a meaning. *Transitions* are not defined explicitly, but instead, a set of *transition rules* define the conditions under which a transition between two global states exists.

Omega UML events

In the case of Omega UML, the semantics involves several types of atomic steps, each defined by a specific transition rule¹. They include: object creation, object destruction, consuming an operation call, starting to fire a state machine transition, executing an assignment, issuing an operation call, returning a result from an operation, returning control from an operation, terminating a state machine transition (i.e., entering a state), and several others.

Each LTS transition (of one of the kinds mentioned above) constitutes an event. The purpose of this work is to set out the framework allowing to use these events as means for interaction between objects.

Event data and meta-data

Every transition (event) in the SOS is defined in a particular context, and depends on a set of elements from the model (the program) and from the run-time that are specifically designated in the *transition rule* inducing it.

We take for example (Figure 3) the transition rule from the semantics of Omega UML which defines an object performing a *triggered operation input* and subsequently firing a state machine transition. (For space reasons, we cannot include the whole SOS definition of Omega UML. The relevant elements of the rule are explained in the sequel, to the extent necessary for understanding the argument.)

The premise of the transition rule contains the contextual elements on which the application of the rule (and hence the existence of the *event*) depends. In particular, the *triggered operation input* event described here depends on:

- The existence *in the model* of a class C , containing a state machine, containing a transition from a state q to a state q' , triggered by an operation op , guarded by a boolean expression g and having as effect a sequence of statements α .

¹ In order to simplify the presentation, we considered here that the semantics of Omega UML is directly defined as SOS. This is in reality not the case, the semantics being given by a set of mapping rules to a different language (IF [4]), for which, in turn, an SOS semantics is defined in [16].

$$\boxed{
\begin{array}{c}
C : q \xrightarrow{[g]op(x)/\alpha} q' \quad \begin{array}{l} \omega \in C \\ \omega.loc = q \\ \omega.v(g) = t \\ \omega.w = op(d).z \end{array} \\
\hline
\Omega \xrightarrow{?op(d)} \Omega \quad \left[\begin{array}{l} \omega.loc \mapsto \alpha \\ \omega.w \mapsto z \\ \omega.v \mapsto \omega.v[x \mapsto d] \end{array} \right]
\end{array}
}$$

Fig. 3. The *triggered operation input* transition rule

- The existence at run-time, *in the global system state* Ω , of an object ω of class C . ω should be precisely in state q and should have a request queue w containing (in front position) a request for op with a data parameter d . Moreover, the value of the guard g under the current valuation v must yield true.

Under these conditions, the transition rule states (in the bottom part) that a *triggered operation input* from the global state Ω is possible, and what is the global state after this event.

It is clear that, if the event is to be used as an interaction mechanism, i.e. it is to be *observed* by some other object, then the (run-time) *data* and the (model) *meta-data* mentioned before characterizes the event and needs also to be observable.

Note that the (meta-)data elements vary from one event type to another. For the operation input shown before, it includes: the concerned object (ω), its class (C), the machine state in which it was before the input (q), etc. For another type of event, for example *object creation*, another set of (meta-)data is relevant: the creator object, the class of the created object, the reference to the new object, etc.

We deduce that every *event* must carry this data as *parameters*, and that the observation construct (which we define below as the language mechanism for manipulating events) must offer access to the parameters. Depending on the degree of *reflectivity* available in the host language, it may or it may not be possible to talk explicitly about the meta parameters.

2.2 Observations

A model execution is a path in the LTS, and therefore corresponds to a sequence of *events* as defined before. An *observation* is a modelling construct which serves to identify and manipulate a sub-sequence of these events that is relevant for a particular goal.

Such a construct is in principle orthogonal to all the other constructs of the *host* language (Omega UML in our case) – in the sense that it serves a different purpose and it is not structurally related (contained into, etc.) to any other existing language construct. Later we will see that it is possible to relate observations to existing constructs (e.g., for specifying a context in which observed events

should occur), but this is not in any way fundamental to the notion of observation and is seen as syntactic sugar.

Observations are used to achieve observation-based interaction between objects (see below in §3), but in order to do so they also have to provide access to the event (meta-)data as outlined in §2.1. This is achieved by considering observations as being objects themselves, with attributes that store the relevant data of the last event matching them at any time during execution. A very good term of comparison from commonly used languages are *exception objects* in languages like C++ or Java: they can be used for communication and for triggering behavior (when they are caught), but they are also objects themselves, with attributes, etc. To continue the comparison, observations can be seen as a sort of exceptions raised by default (when a matching event occurs), which can be “caught” and thus trigger some behavior, but which do not have the disruptive quality of “usual” exceptions if they are not caught.

We argue that all *observations* that are interesting for modelling can be constructed based on a limited number of *elementary observations* (roughly corresponding to the transition rules of the SOS) and on a set of *operations on observations* which are derived from the most common set operations (union, intersection, complementing, projection).

Elementary observations

We stated before that events correspond to transitions from the semantic LTS. As such, each event has a *kind*, which is the *transition rule* of the SOS from which it is derived. It follows logically that we can define elementary *observations* which correspond to these event *kinds*. If the sequence of events generated by an execution is $\mathcal{E} = (e_1, e_2, \dots)$, then an *elementary observation* \mathcal{O} generates a sub-sequence $\mathcal{E}_{\mathcal{O}} = (e_{k_1}, e_{k_2}, \dots)$ such that all e_{k_i} are of kind \mathcal{O} .

In the case of Omega UML, for example, to the *triggered operation input* rule in Figure 3 corresponds an elementary observation $\mathcal{O}_{toinput}$. Note that the number of rules is usually quite small even for complex languages (around 15 for Omega UML depending on how the semantics is defined), and that all types of events are not necessarily interesting to observe.

Observation operators

Remember that the semantic model of an observation is a subset of the whole sequence of events generated in a system execution (\mathcal{E}). It is then natural to allow observations to be composed using set-based operators: union, intersection, complementation, projection.

The semantics of observation union, intersection and complementation is self-explanatory. The projection operation allows to obtain from an observation \mathcal{O} (with semantic model $\mathcal{E}_{\mathcal{O}}$) another observation \mathcal{O}' whose semantic model $\mathcal{E}_{\mathcal{O}'}$ is a subsequence of $\mathcal{E}_{\mathcal{O}}$, based on a boolean condition on the event parameters.

Here are some examples of use of the operators (for simplicity, we use the standard mathematical notations for observation operators):

- for identifying the triggered operation inputs that affect only a particular variable x one can take a projection of the elementary observation $\mathcal{O}_{toinput}$ (suppose the *.store* meta-data is a list of the attributes affected by an input):

$$\mathcal{O}_{toinput(x)} = \mathcal{O}_{toinput} \Big|_{x \in .store}$$

- for identifying the events that lead to modifying a particular variable x , one can take the union of $\mathcal{O}_{toinput(x)}$ (defined before) and of the observation of *assignments* which affect x :

$$\mathcal{O}_{mod x} = \mathcal{O}_{toinput(x)} \cup \mathcal{O}_{assign} \Big|_{x \in .store}$$

Implicit projections

Often one is interested in observing a particular kind of events only in a particular context (e.g., the *inputs* performed by one particular object). In principle, the projection operator presented before is sufficient for specifying the context, although in practice this may be syntactically awkward (for our example, one needs to explicitly designate the identity of the concerned object, raising the question of how this identity is obtained).

Therefore, it may sometimes be useful to define observations in the program not as top-level constructs but within a context (e.g., inside a class), meaning that there is an implicit projection of the observation based on the context. This does not augment the expressive power of observations defined before, and is only syntactic sugar.

3 Behavior composition using observations

The purpose of introducing *observations* is to be able to identify *events* and use them in the behavioral specification of components (objects in Omega UML). The way to do this depends on the computational model assumed by the host language. In the following, we propose a solution which works for Omega UML and can presumably be adapted to any concurrent language (synchronous or asynchronous).

The elementary extension that is necessary is a statement which, for a given observation o , waits until an event matching o is produced. Let $wait(o)$ denote this statement. Since this statement is possibly blocking, it has to be introduced in a way which is consistent to the language definition: for example in Omega UML the assumption is that the only blocking statements are the *transition triggers*. Therefore, $wait(o)$ has to be defined as a new type of transition trigger.

Semantics of observation waiting

Clarifying the semantics of the $wait(o)$ trigger mainly means answering the following question: after an event e that matches o has occurred, when (and for how long) is $wait(o)$ enabled (i.e., able to fire the associated transition).

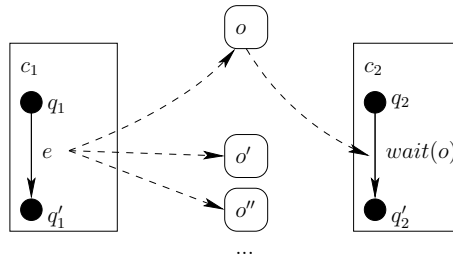


Fig. 4. Observation-based communication

When a component (object) c_1 executes an action which induces an event e matching an observation o which is waited upon by a second component c_2 (see Figure 4) this implies an *implicit communication* taking place between c_1 and c_2 . Prior to this communication, the attributes of o are updated (with the event data). It is a semantic choice whether to consider this communication *synchronous* or *asynchronous*. In the former case, the *wait* transition can be fired only if c_2 is already in state q_2 when e occurs. c_2 becomes executable immediately (in terms of threads, it goes from suspended to executable). In the latter, the *wait* transition may be fired at a later moment, including after c_2 has executed some other actions, and it implies that the run-time has the ability to keep the memory of the event for every component until it is able to “consume” (observe) it.

Unlike for normal message-based communication, where asynchronous and synchronous messages have the same expressive power (i.e. each one can be expressed with the other, see for ex. [5]), for observations the synchronous semantics is *strictly more powerful* in the following sense: given an asynchronous observation mechanism, it is impossible to express a synchronous observation without modifying the specification of the event *source* component (c_1 above). (An explicit waiting for an acknowledgement has to be inserted in c_1 in order to achieve synchrony.) Since the reason of existence of the observation-based interaction is precisely to make possible for c_2 to react to events *implicitly* produced by c_1 , requiring modifications in c_1 in order for it to work would be self-defeating.

In consequence, we settle for the choice of *synchronous* observations. For space reasons, we refrain from giving the SOS rules defining event production and observation in Omega UML. However, the precise semantics can be understood by comparison to known synchronization constructs from other frameworks, in particular to *condition variables* from Hoare’s monitors [11] (also known from the Java language via the `wait` and `notifyAll` functions of `java.lang.Object`). The semantics of $wait(o)$ is identical to waiting on a *condition variable* associated to o . Event generation is the (atomic execution of) updating o ’s attributes with the actual event parameters, followed by signaling the condition variables associated to all matching observations (o, o', o'' in Fig. 4).

We note that based on the synchronous semantics it is possible to achieve stronger constraints if necessary: for example if, upon e , c_2 has to take a series of actions *before* c_1 may continue, this can be achieved simply by giving c_2 a higher priority compared to c_1 . To do so, one may use either the capabilities of the run-

time (priority levels are available for example in all real-time operating systems), or constructs available in the host language (e.g., *dynamic priority rules* as they exist in Omega UML [9]). Although this kind of combination of language features is important, we cannot focus more on it in this paper.

4 Observers as dynamic aspects

Aspect orientation [15] is based on the idea that, in the process of designing a (software) system, certain concerns or requirements may impact several components of the established system architecture. Such concerns are called *cross-cutting aspects*. Aspect-oriented languages usually consist of a traditional (object oriented, procedural or functional) host language for specifying the architecture and the basic system functionality, and of a set of constructs for specifying aspects.

An aspect is generally defined in terms of the aspect’s *joinpoints* – the program elements that are affected by the aspect and the conditions under which this happens, and the aspect’s *advice* – the structural and/or behavioral elements added by the aspect. The semantics of an aspect-oriented language is given by the rules for composing (*weaving*) aspects and the basic system specification into an executable model.

The *observation* constructs defined in the previous sections may be used as a basis for defining joinpoints: the execution of aspect-related code is then triggered by the occurrence of events. By combining this with the control-flow constructs already available in a host language such as Omega UML, aspects can be triggered not only by individual events but by arbitrary patterns of events, with arbitrary conditions based on event data, etc. This is in general more expressive than what can be achieved in most “classical” aspect-oriented models, like the one of AspectJ [14], in which *joinpoints* are based mostly on syntactic conditions.

We illustrate the approach with a commonly used example, which adds a *logging* functionality to an existing system – in our case the ATM described in §1.1. Our logging differs from more classical examples in that it must take place under some run-time conditions which involve the occurrence of certain events in a specific order. While remaining fairly simple, this example suggests what can be achieved by combining observations with full fledged state machines.

We consider the following requirement: each time the cash unit successfully locks an amount of cash greater than 100 for delivery, the ATM must subsequently write a log entry with the outcome of the transaction (whether it was canceled or whether the cash was effectively delivered). Fig. 5 shows the specification of this aspect; for simplicity, we use the mathematical notation introduced in §2.2, the actual model-based notation differing from it only syntactically.

The aspect is implemented as a separate active object, with its own state machine. It uses the following observations: \mathcal{O}_{ctl} matching the calls to `TryLockCash` with an amount higher than 100, constructed from the elementary observation \mathcal{O}_{pocall} which denotes all calls to primitive operations. $\mathcal{O}_{rtltrue}$ and $\mathcal{O}_{rtlfalse}$ matching the return from `TryLockCash` successfully, respectively unsuccessfully (constructed from the elementary observation $\mathcal{O}_{poreturn}$ which denotes all primitive operation returns). $\mathcal{O}_{deliver}$ and \mathcal{O}_{cancel} denote the triggered operation inputs

$$\begin{array}{l}
\mathcal{O}_{ctl} = \mathcal{O}_{pocall} \mid \begin{array}{l} .op = \text{CashUnit}::\text{TrylockCash} \wedge \\ .pars[0] \geq 100 \end{array} \\
\mathcal{O}_{rtltrue} = \mathcal{O}_{poreturn} \mid \begin{array}{l} .op = \text{CashUnit}::\text{TrylockCash} \wedge \\ .ret = \text{true} \end{array} \\
\mathcal{O}_{rtlfalse} = \mathcal{O}_{poreturn} \mid \begin{array}{l} .op = \text{CashUnit}::\text{TrylockCash} \wedge \\ .ret = \text{false} \end{array} \\
\mathcal{O}_{deliver} = \mathcal{O}_{toinput} \mid .op = \text{CashUnit}::\text{DeliverCash} \\
\mathcal{O}_{cancel} = \mathcal{O}_{toinput} \mid .op = \text{CashUnit}::\text{CancelDelivery} \\
\mathcal{O}_{dc} = \mathcal{O}_{deliver} \cup \mathcal{O}_{cancel}
\end{array}$$

state machine logger

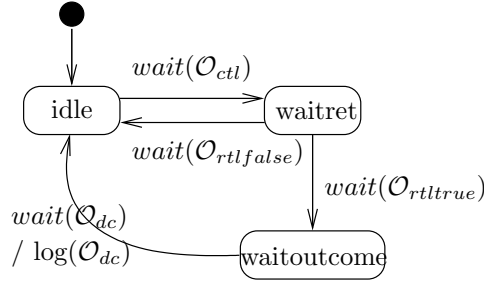


Fig. 5. A logger aspect based on observations.

for **DeliverCash** and respectively **CancelDelivery**, and \mathcal{O}_{dc} is the union of the two.

Based on these, the control structure of the aspect is as follows: whenever it observes a call to **TrylockCash** with an amount higher than 100 and which returns **true**, it waits for an input of either **DeliverCash** or **CancelDelivery** (\mathcal{O}_{dc}) and when this occurs it logs the related information (which can be retrieved directly from \mathcal{O}_{dc}).

We have applied this form of aspects to model other cross-cutting concerns, such as *alarm management* in an embedded controller. Alarm management involves dynamic conditions on the order of events, their data and also their timing (e.g., an alarm has to be issued if the plant does not reach some operating state within n milliseconds from a *start* event). The construction is similar (if more complex) to the one in the logger example.

Finally, we note that with this approach, nothing distinguishes an aspect from a “normal” system component, and one can mix observation-based behavior and conventional communication-based behavior inside the same component. This opens the way to a methodology of system specification in which well-behaved (decomposable) concerns are implemented across the component architecture by conventional means, while ill-behaved (cross-cutting) concerns are implemented using observations. Such a methodology would approach the goals put forward by Jacobson and Ng [12] in their *use-case slicing* method, which has stimulated much interest in recent years, but for which current aspect specification frameworks are generally considered not powerful enough.

5 Related work and conclusions

The concepts of event and observation presented here are an evolution of those we described in [9]. There they served as basis for defining a particular type of aspects (execution timing constraints) by using a dedicated *declarative* language, which is less flexible than the generalized use of observations as interaction mechanism, proposed here.

A criticism against observation-based interaction is that it appears to lead to tightly coupled component architectures. The fact is that implementing cross-cutting concerns inherently leads to tightly coupled components, and aspect-oriented languages aim at reducing the burden of the designer by making part of the coupling *implicit*. Our approach does the same: when the coupling between two components is unidirectional (observed-observer), it provides a way to avoid the impact on the specification of the observed component.

The body of literature dedicated to aspect models and languages is wide and growing rapidly. An important classification criterion for aspect frameworks concerns the constructs for defining *join points* and their expressive power. From this point of view, most of the proposed frameworks (including that of AspectJ [14]) use syntactic joinpoint models. However, a few recent proposals are closer to ours, in that they propose event (trace) based joinpoint models [20, 2, 8], sometimes combined with a form of dynamic (run-time) weaving (e.g., [7, 17]).

In [20, 2, 17], join points (sometimes called *tracecuts* because they are based on a multi-event trace) are defined in a declarative way, for example as a regular expression on events, and concern only sequential programs. The computational model that is closest to ours is the Concurrent Event-Based AOP (CEAOP) defined in [7], and which is based on the same principles of parallel composition of system components and aspects, and of event based synchronization. The main difference is that we define the characteristics of events and the observation construct (§2), whereas in [7] events are just simple synchronization labels. Also, in [7] the authors distinguish normal system components and aspects (although their computation model, like ours, does not require this distinction). We think that not making the distinction, and uniformly offering the same observation mechanisms to all components can lead to interesting results in keeping concerns related to each use-case separate inside the component descriptions (this technique is known as *use case slicing* [12]). Further evidence is needed to support this claim.

Still concerning future work, we note that the observation concept (in particular, access to event meta-data) requires reflective support in the language in order to be fully satisfactory. Therefore, we plan to test the concepts presented here on a language which offers such support, together with support for asynchronous concurrency, such as some extensions of Python.

The observation mechanisms we originally proposed in [9] had been implemented in a centralized way, in the execution platform. This is possible also in the present setting and it is acceptable since the Omega profile is mostly dedicated to simulation and formal property verification, where a centralized event monitor does not affect performance. If the target is final implementation, event observation must be done more efficiently, e.g. by code instrumentation based on

a publish-subscribe mechanism. The work of [7], on a similar model, shows that this is in principle possible.

References

1. B. Algayres, Y. Lejeune, and F. Hugonnet. GOAL: Observing SDL behaviors with ObjectGEODE. In R. Braek and A. Sarma, editors, *SDL'95: with MSC in CASE, Proceedings of the 7th SDL Forum*. Elsevier Science B.V., 1995.
2. Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie J. Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to aspectj. In Ralph Johnson and Richard P. Gabriel, editors, *OOPSLA*, pages 345–364. ACM, 2005.
3. Marius Bozga, Susanne Graf, and Laurent Mounier. If-2.0: A validation environment for component-based real-time systems. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 343–348. Springer, 2002.
4. Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. The if toolset. In Marco Bernardo and Flavio Corradini, editors, *SFM*, volume 3185 of *Lecture Notes in Computer Science*, pages 237–267. Springer, 2004.
5. Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages (Third Edition)*. Addison Wesley, 2001.
6. Werner Damm, Bernhard Josko, Amir Pnueli, and Angelika Votintseva. Understanding uml: A formal semantics of concurrency and communication in real-time UML. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 2852 of *Lecture Notes in Computer Science*, pages 71–98. Springer, 2002.
7. Rémi Douence, Didier Le Botlan, Jacques Noyé, and Mario Südholt. Concurrent aspects. In Stan Jarzabek, Douglas C. Schmidt, and Todd L. Veldhuizen, editors, *GPCE*, pages 79–88. ACM, 2006.
8. Robert E. Filman and Klaus Havelund. Realizing aspects by transforming for events. Technical Report 02.05, RIACS, September 2002. Presented at ASE'2002 Workshop on Declarative Meta-Programming.
9. Susanne Graf, Ileana Ober, and Iulian Ober. A real-time profile for UML. *STTT*, 8(2):113–127, 2006.
10. David Harel and Hillel Kugler. The rhapsody semantics of statecharts (or, on the executable core of the UML) - preliminary version. In Hartmut Ehrig, Werner Damm, Jörg Desel, Martin Große-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkämper, editors, *SoftSpez Final Report*, volume 3147 of *Lecture Notes in Computer Science*, pages 325–354. Springer, 2004.
11. C. A. R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.
12. Ivar Jacobson and Pan-Wei Ng. *Aspect-Oriented Software Development with Use Cases*. Object Technology Series. Addison-Wesley, 2005.
13. Claude Jard, Jean-François Monin, and Roland Groz. Development of Véda, a prototyping tool for distributed algorithms. *IEEE Trans. Software Eng.*, 14(3):339–352, 1988.
14. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In Jørgen Lindskov Knudsen, editor, *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

15. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.
16. Yassine Lakhnech Marius Bozga. IF-2.0 common language operational semantics. Technical report, VERIMAG, September 2002. Available at <http://www-if.imag.fr>.
17. Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée. Explicitly distributed AOP using AWED. In Robert E. Filman, editor, *AOSD*, pages 51–62. ACM, 2006.
18. Object Management Group. *Unified Modeling Language*. Available at <http://www.omg.org/spec/UML/>
19. Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.
20. Robert J. Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In Richard N. Taylor and Matthew B. Dwyer, editors, *SIGSOFT FSE*, pages 159–169. ACM, 2004.