Behavioral Modelling and Composition of Object Slices Using Event Observation

Iulian Ober[1], Bernard Coulette[1], and Younes Lakhrissi[1,2]

[1] Université de Toulouse - IRIT
118 Route de Narbonne, 31062 Toulouse, France
{iulian.ober,bernard.coulette,lakhriss}@irit.fr
[2] ACSYS, Faculté des Sciences de Rabat

Abstract. Some analysis and design methods for complex software systems lead to the specification of components (classes) by *slices*. It is the case of the use-case slicing technique proposed by Jacobson and Ng, and of view-based modelling proposed by Nassar et al. The composition of class slices is known from the literature to be closer to aspect composition than to traditional interface-based composition, but remains largely an open problem.

In this paper we propose a set of constructs to support the behavioral specification and composition of class slices, based on the idea of non-intrusive *event observation*. This allows slices to be specified separately – for example by different design teams – and to be integrated later without changes. The proposal is made in the context of VxUML, a language which supports view-based and use-case-driven separation of concerns.

1 Introduction

When tackling the complexity of large software systems, separation of concerns is essential for keeping the development process, the produced models and the code manageable. The separation of concerns can be done in different ways, but the objectives are always the same: being able to identify relatively independent "parts", so that they can be distributed to different actors of the process, be designed and built independently and, at the end, be integrated with the least possible effort and in a way which allows for future maintenance and evolution.

Traditional software decomposition methods rely upon a notion of *component*. A component is an entity which fulfills one or more functions in the overall system architecture and provides access to these functions via some form of *interface*. In general one end-user functional requirement is fulfilled by the collaboration of several components, and one component participates in fulfilling several requirements. In some extreme cases, small bits of functionality related to one requirement are spread over a very large number of components (a situation called *scattering*), and some components contain, beside their core functionality, bits of scattered functionality tightly interwoven (*tangling*). Such examples provided the motivation for aspect oriented programming [12].

K. Czarnecki et al. (Eds.): MoDELS 2008, LNCS 5301, pp. 219–233, 2008.

Recently there have been proposals for methods which aim to keep different concerns about end-user functional requirements separate throughout analysis, design and implementation. In general, this is done by superimposing a structure for functional decomposition over a (sufficiently resilient) component architecture. In this work, we concentrate on two related proposals: the use-case slicing approach of [9] and the view-driven approach of [15]. As we show in Section 2, both methods lead to designing objects (classes) as a composition of *slices*.

The subject of this work is the behavioral specification and composition of such object slices. The problem is known (see for ex. [9]) to be closer to aspect composition than to traditional interface-based composition. We propose a new behavior specification construct based on *event observation* and we show that this leads to good results in terms of slice coupling and support for incremental design (addition of slices).

The level of specification that we aim at is that of detailed, executable design models. We focus in particular on design models where state machines are used as operational behavior specifications for objects, something that is common in the design of reactive systems (e.g., embedded software, protocol layers, etc.). We integrate our proposal in VxUML, an executable UML profile which supports operational state machines and class slices (along the lines of [15]).

The paper is structured as follows: Section 2 discusses the motivation for this work, which comes from the necessity of flexible mechanisms for specifying and composing use-case slices and view-based slices. Section 3 describes the central concepts of our proposal: *events* and *probes*. Section 4 discusses (on an example) how these concepts are used for object slice specification and composition. We end the paper by comparing our proposal to some existing approaches (Section 5), and by drawing conclusions and the main lines of future work.

2 Object Slices: Background and Open Issues

In this section we briefly introduce the two design methods that we consider in this work, which both lead to the specification of classes as a composition of slices. We use as running example a hotel management system (the example also used throughout [9]). For brevity, we do not introduce the overall architecture and the whole set of requirements; we concentrate instead on the relevant aspects of one class, *Room*, which is involved in the realization of most of this system's use cases (reservation, customer check-in, check-out, maintenance, etc.)

We consider in particular the case of reactive objects, for which the behavior of slices is defined using state machines. Because slices can be cohesively attached to each-other, this leads to problems that are discussed at the end of the section on a motivating example.

2.1 Use-Case Slicing

Use-case slicing is an analysis and design discipline introduced in [9]. Its main prescription is that, for every identified use case of a system and for every component (class) participating in it, the aspects of the component pertaining to

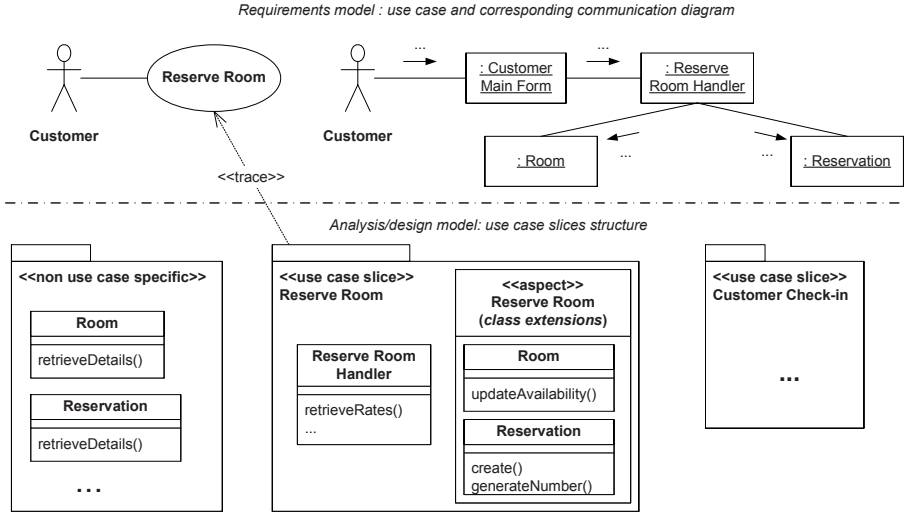

Fig. 1. Packaging of use case slices (as recommended in [9])

that use case are to be kept separate from the aspects pertaining to other use cases and from the aspects that are not use-case-specific. This prescription holds for analysis, design and implementation components. A *use-case slice* is a modelling element which contains the aspects specific to one use case, from all the involved classes.

Only the high-level principles for packaging slices are described in [9]. Figure 1 (extracted from the hotel management example) illustrates the main lines of the approach. For every use case identified in the requirements, there is a corresponding package in the analysis (and in the design) model, stereotyped as `<<use case slice>>`, containing the use-case-specific parts of every involved class (we call these *class slices*). Additionally, the analysis (design) model can contain component specifications limited to their non-use-case-specific features. Dependency relations (`<<trace>>`) allow to trace back from every slice to the originating use case from the requirements model.

The precise behavior specification and composition mechanisms for use-case slices are not further detailed in [9]. Moreover, executable object behavior descriptions based on state machines are not explicitly addressed. State machines are mentioned as a means to analyze object lifecycle, but they are left aside in detailed design.

2.2 View-Based Slicing

View-based slicing is an analysis and design discipline defined in [15]. It is based on the idea that a complex system usually has many external actors interacting with it, and that the concerns of the different actors can in general be separated and handled independently throughout system analysis and design.

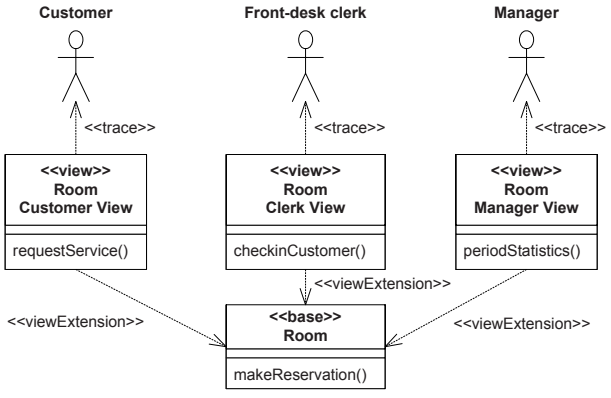

Fig. 2. Structure of a multi-view class in VUML [15]

While the separation lines are different from those recommended in use-case slicing, the principle of constructing each component (class) as a composition of slices containing distinct functionality remains the same. This is visible in the structure of the models (see the example in Figure 2): each *multi-view* class is specified via a *base* (grouping functionality that is not specific to any actor) and a set of actor-specific slices called *views*.

In the original view-based approach described in [15], only behavior specifications based on operations are discussed. Views are assumed to contain operations dedicated to a specific actor. This implies that the functionality captured in different views is strictly orthogonal, which is not always the case: the requests from different actors impact one another in general. Like for use-case slices, operational state machine specifications were not explicitly addressed.

2.3 State Machines: Semantic Choices

As we noted before, we are interested in the case where the complete executable specification of objects (slices) is given in terms of UML state machines. The UML standard [18] explicitly leaves open certain questions concerning the semantics of state machines, like *the concurrency model* (i.e. the semantics of active/passive objects, their relation to control threads), the precise semantics of *object interaction primitives* (how calls are handled by active objects, what is the relation between calls and state machine, etc.) or the *concrete syntax of actions* (used in transition effects and method bodies).

In order to have a semantically well-founded proposal, we have to make choices with respect to the aforementioned UML semantic variation points. In previous work we have participated to the definition of an executable profile for UML called Omega UML [6], for which we have developed execution and simulation tools [17]. The semantics of Omega UML is appropriate for the concepts proposed in this paper and convenient for rapid prototyping. Therefore we decided to integrate the same semantics into the VUML profile [15], resulting in an executable version of it to which we refer as VxUML.

For a complete definition of the Omega profile, we refer the reader to [6, 17]. Let us only note that the Omega concurrency and communication model corresponds largely to the semantics of UML in the Rhapsody tool [7] and that the concrete action syntax relies on conventions widely used in imperative object oriented languages (for example in Ada 2005).

2.4 Slices and State Machines: Problem and Motivating Example

As we already remarked, slices are closer to aspects than to stand-alone objects. This is because objects are loosely connected (in good designs), while slices are by construction tightly connected. A slice that extends the functionality of another slice cannot in general be specified based only on a black-box description of the extended slice, as it has to plug in at specific points of its behavior. However, it is desirable to be able to develop such slices independently, and in a way that allows integrating them without modifying the extended slice.

In the context of the hotel management system, we consider the example of a new view (and a corresponding use-case) being added to the requirements: the *room maintenance* view. We place the requirements on it:

- The maintenance can be triggered by various conditions (specific request from the manager or the front-desk clerk, elapse of a period, etc.). All the conditions are not specified a priori and it should be easy to add new ones.
- If the maintenance is triggered while the *Room* is unoccupied, a thorough check is performed.
- If the maintenance is triggered while the *Room* is occupied, maintenance is put on hold. If after N time units (days) the room is still occupied, a lightweight maintenance check is performed. A thorough check is performed only when the customer checks out.

These requirements can be realized by adding a *Room Maintenance* slice to the *Room* class. Figure 3 shows an informal analysis model of the state machine for *Room Maintenance*. The actual maintenance procedures are not detailed further.

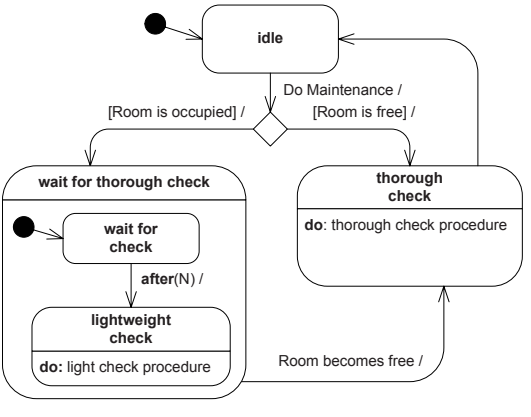

Fig. 3. Analysis state machine of the *Room Maintenance* slice

Explicit message-based communication is ill-adapted to specifying this type of slice: it implies modifying the other slices of *Room* (and possibly other classes) to introduce explicit calls for requesting maintenance (`Do Maintenance` trigger) and for signaling when the `Room becomes free`, as well as to ensure that check-in is delayed while the *Room* is in `thorough check`.

Our objective is to define the mechanisms allowing to write a self-contained executable specification of this slice without modifying the rest of the system. The idea is to let the slice specify precisely what *events* from the rest of the system it needs to *observe*, and how it reacts to them.

3 Behavior Specification Based on Event Observation

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.
- *probe*: a modelling 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).

While we introduce these notions in the context of VxUML, their definition is sufficiently generic so they can be appended to any language provided with a *structural operational semantics* (SOS, in the sense of [19]).

3.1 Events

We relate *events* to the smallest (i.e. indivisible) state changes described by the semantics of a model/language. In case of a structural operational semantics, events correspond to the transitions of the semantic *labeled transition system* (LTS) associated with a model.

SOS terminology. In SOS[19] 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 a global state identified as *initial* represent the possible *executions* of the program.

For a concurrent object-oriented language like VxUML, a global state includes the attribute values, states and request queues 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.

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

Fig. 4. The *triggered operation input* transition rule

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

VxUML Events. In the case of VxUML, the semantics involves several types of atomic steps, each defined by a specific transition rule[1]. 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 (entering a state), and several others. Each LTS transition of one of the kinds mentioned above constitutes an event.

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 4) the VxUML transition rule which defines an object performing a *triggered operation input* and subsequently firing a state machine transition. (For brevity, we do not include the whole VxUML SOS. Only the relevant elements of the rule are explained here, 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 *event*) depends. In particular, the *triggered operation input* event described here depends on:

– The existence of a class C *in the model*, containing a state machine, containing a transition from a state q to a state q', triggered by an operation

[1] In order to simplify the presentation, we consider here that the semantics of VxUML is directly defined as an SOS. This is in reality not the case, the semantics being given by a set of mapping rules to a different language (IF [3]), for which, in turn, SOS semantics is defined in [13].

op, guarded by a boolean expression g and having as effect a sequence of statements α. (This is the meaning of $C : q \xrightarrow{[g]op(x)/\alpha} q'$.)
- The existence at run-time, *in the global system state* Ω, of an object ω of class C ($\omega \in C$). ω should be precisely in state q ($\omega.loc = q$) and should have a request queue w containing in front position a request for *op* with a data parameter d ($\omega.w = op(d).z$). Moreover, the value of the guard g under the current attributes valuation v must yield true ($\omega.v(g) = t$).

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 the global state after this event is.

If the event is used as an interaction mechanism, i.e. it is *observed* by some other object, then the (model) *meta-data* and the (run-time system state) *data* mentioned above 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 a different type of event, for example *object creation*, a different set of (meta-)data is relevant: the creator object, the class and reference of the created object, etc.

This context data constitutes what we call the *event parameters*. The *probe* construct, defined below as the language mechanism for manipulating events, offers access to these parameters.

3.2 Probes

A *probe* is a modelling construct which serves to identify and manipulate events that are relevant for a particular goal. In particular, we use probes for modelling implicit interaction between objects (see Section 3.3).

Analogy to Exception Objects. Probes must provide access to the event data and meta-data as outlined in Section 3.1. This is achieved by considering probes as being *objects* themselves. Their attributes store the relevant data of the last event matching them at any time during execution. A very good analogy from commonly used languages are *implicitly raised exception objects* in C++ or Java: they can be used for communication and for triggering behavior (when they are "caught"), and they are also objects themselves, with attributes carrying information about the context in which they were raised. The difference is that the activation of a probe by an event does not disrupt the normal execution if not used ("caught").

Elementary Probes. We argue that all interesting *probes* can be constructed based on (1) a limited number of *elementary probes* (roughly corresponding to the transition rules of the SOS) and (2) a set of *operations* derived from common set operations (union, intersection, complementing, projection).

We saw 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. We define *elementary probes* to correspond to these event *kinds*.

Since a model execution is a path $\mathcal{E} = (e_1, e_2, ...)$ in the semantic LTS, i.e. a sequence of events, a probe corresponds to a sub-sequence of these events. An *elementary probe* $\mathcal{P}$ corresponds to a *sub-sequence* $\mathcal{E}_\mathcal{P} = (e_{k_1}, e_{k_2}, ...)$ such that all e_{k_i} are of kind $\mathcal{P}$.

For example, in the case of VxUML, there will be an elementary probe $\mathcal{P}_{toinput}$ corresponding to the *triggered operation input* rule in Figure 4. This probe can be used to observe every event of type *triggered operation input*. We note that the number of SOS transition rules (and hence, the number of elementary probes) is usually quite small even for complex languages (around 15 for VxUML depending on how the semantics is defined).

Probe Operators. The semantic model of a probe is a *time-ordered sub-sequence* of the whole sequence of events generated in a system execution ($\mathcal{E}$). It is natural to allow probes to be composed using sequence operators: union, intersection, complement, projection.

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

Hereafter are two examples using the operators (for simplicity, we use the standard mathematical notations for probe operators):

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

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

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

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

Implicit Projections. In principle, the probe construct is orthogonal to all the other constructs of VxUML – in the sense that it serves a different purpose and it is not structurally related (contained into, etc.) to any other existing language construct. Therefore, in our model probes are defined as top-level constructs, on par with classes.

However, 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 referred to in the probe definition).

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

3.3 Behavior Specification and Composition with Probes

The purpose of introducing *probes* is to be able to identify *events* and use them for the behavioral specification of objects (slices). This is done by a construct allowing an object (slice) to *wait* for the arrival of an event e matching a probe $\mathcal{P}$. The natural place for such a construct is in the trigger of state machine transitions – therefore $wait(\mathcal{P})$ is defined as a new type of trigger.

Defining the semantics of *probe update* (i.e. the updating of $\mathcal{P}$'s attributes upon arrival of e) and *activation* (i.e. the triggering of a $wait(\mathcal{P})$ transition) implies making some choices. Among others, we have to decide whether the implicit communication that takes place between the object (slice) that produces the event and the one that *waits* for it is asynchronous (i.e. there is a memory of the activation status) or synchronous.

The semantics that we chose in VxUML is fully synchronous: e, the *update* of o, and the *activation* of subsequent transitions are executed atomically. In this semantics, the *wait* trigger works similarly to *condition variables* from Hoare's monitors [8] (also known from Java's `wait` and `notifyAll` primitives). For space reasons we refrain from further motivating our choices and from giving the complete SOS semantics.

3.4 UML Representation

This is a brief overview of the conventions employed for representing probes in VxUML (see examples in the next section):

- A probe defined in the context of a classifier is represented by a UML *Property* stereotyped as `<<probe>>`. If the probe is global (defined at top level) and there is no top level "system" class in the model, then the probe may be represented by an UML *InstanceSpecification*.

  In both cases, a probe is characterized by a name and a type. The type is either an elementary probe type like `OperationCallProbe`, or the abstract type `Probe` – when the probe may match events of different types, e.g., in case of unions.
- When the probe is defined by an expression with set operators, the expression is modeled as a constraint attached to the *Property* (or *InstanceSpecification*); in particular, the syntax for projection expressions is "`when` *boolean-condition*".
- A probe declaration can be placed inside a class (slice), meaning that there is one actual probe per instance of the class. In this case, the keyword `this` is used in the projection condition to denote the corresponding instance.

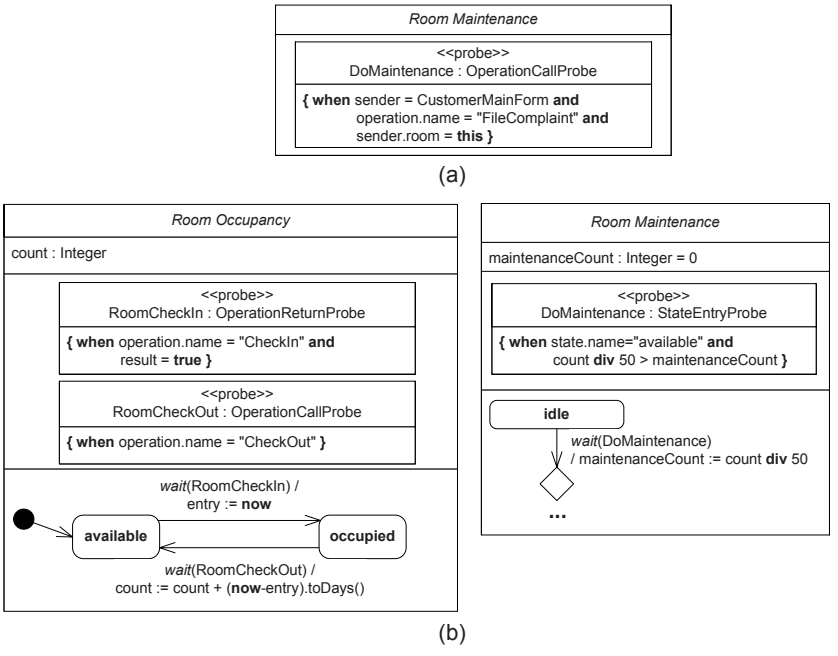

Fig. 5. Specification of the *Room Maintenance* slice with probes

4 Solution for Composition of Object Slices

The *probe* construct defined in the previous section is well-adapted for specifying use-case or view-based slices, as it allows to isolate the specification of the tight coupling between object slices in a way that the behavior specification of the slices themselves is not cluttered. We examine this on the *Room maintenance* example introduced in Section 2. As it was mentioned, the main problem is linking the behavior of this slice with that of other slices, in particular for triggering the `Do Maintenance` transition. We consider two examples:

- Room maintenance is triggered by a single event: whenever the Customer lodged in the Room files a complaint. In this case, one can simply use a `DoMaintenance` probe which monitors the operation call representing this action. Figure 5 (a) shows the declaration of the probe.
- Room maintenance is triggered by a more complex condition: whenever the *Room* is left by a customer, and it has been occupied for more than 50 days since the last maintenance. The test for this condition can be either embedded in the *Room Maintenance* slice, or it can be isolated in a separate slice specifically created for this purpose. In any case, the other slices need not be modified.

  Figure 5(b) shows the model corresponding to the second alternative. The new slice, called *Room Occupancy*, uses two *probes* for customer check-in and

check-out, and counts the total occupation days in the `count` variable. The *Room Maintenance* slice, via the `DoMaintenance` probe, monitors whether the total occupation grows past a multiple of 50 when a customer checks out.

Note that in both examples, the executable (design) state machine of *Room Maintenance* slice remains very close to the analysis model from Figure 3 (the overall structure is the same, only the "informal" transition triggers are refined to probe triggers).

As it was mentioned in Section 2, one of the reasons for designing systems in slices is to be able to distribute the design tasks to independent actors/teams. Our experience shows that this type of process is well supported, since the slices may in general be designed based on undefined probes in a first time (using the abstract *Probe* type). Integration is done at the end by concretizing the definitions of probes.

A Note on Changing Requirements. In real projects requirements often change on the fly, and the design evolves with them. This means that slices can lately be added to a model, but also that the specification of existing slices may change over time (by adding new states, changing state names, adding transitions, etc.).

Probes provide support for the late definition of interfaces and bindings between object and/or slices (in the extreme case, at the very end of the development cycle, as suggested above). This form of modularity provides *isolation* in case of changing specifications. This means that the impact of a slice change on the rest of the system is often reduced to changing the definitions of the probes with which it interacts.

A Note on Spread Aspects. The *Room Maintenance* example is prototypical for what occurs in "localized" slices, i.e. slices that realize a functionality involving only a few objects (this is the case of functional use-case slices from [9] or of view-based slices).

In typical examples for aspect oriented programming, a functionality (usually an infrastructure-related one, like logging or access control) may be spread over a multitude of objects. Although there is no place for a full example, we maintain that our framework can be very effective for specifying such aspects. A logging functionality for example can be concentrated in a unique *logger* object, which observes all the relevant events throughout the system. In this case, probes are used for implicit communication between objects, instead of slices.

5 Related Work and Comparisons

Observation-Driven Specification. In [6] we defined an event-based framework for specifying execution timing constraints, using a dedicated *declarative* language. This work is a distant evolution of the ideas from [6], as the notions of event and probe defined here allow a generalized use of observation as object (slice) interaction mechanism.

Observation-driven behavior specification has to our knowledge not been proposed previously in the context of object-oriented modelling and programming languages. We took some inspiration in the concept of *observer*, introduced in [10] as a property specification formalism (and re-used in several other verification tools). Related ideas can also be found, under very different forms, in other domains like system tracing and debugging (see for example DTrace and the D language [14]) or autonomous agents.

Aspect-Oriented Programming. Although targeting the particular problem of designing systems by slices (view or use-case driven), our work has to be compared with results from the area of aspect-oriented modelling and programming. Indeed, designing systems by slices, like realizing cross-cutting concerns, inherently leads to tightly coupled components. Similarly to aspect-oriented frameworks, we are trying to cope with this by making part of the coupling *implicit*.

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 *joinpoints* and their expressive power. In our framework, by combining event observation with the control structure of state machines, 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 [11], in which *joinpoints* are based mostly on syntactic conditions. However, a few recent proposals are closer to ours, in that they propose event (trace) based joinpoint models [20, 1], sometimes combined with a form of dynamic (run-time) weaving (e.g., [5, 16]).

In [20, 1, 16], joinpoints (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 [5], 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 probe construct, whereas in [5] events are just simple (uninterpreted) synchronization labels.

Aspects for state machine-based specifications are relatively less studied than their operation-oriented counterparts. SDMATA [21] is a transformation-based framework in which joinpoints are specified as state diagram patterns, and advices are specified as operations (element addition/deletion) on these patterns. The latest versions of the Motorola Weavr [4] introduce a similar framework, with a joinpoint model based on the structure of state machine diagrams and the possibility to modify this structure in aspects. Writing pattern-based aspects requires more than common programming skills and some knowledge of the structure of the UML state machines metamodel. Also, in our framework aspects are *interfaced* with the rest of the system based on a clearly determined set of *observable events*, whereas in SDMATA and in Veavr aspects are written in terms of the *structure* (i.e. *implementation*) of the state machines on which they are superposed. For this reason, we expect our observation-based aspects

to be somewhat less sensitive to implementation changes in the rest of the system. On the negative side, these frameworks allow writing more intrusive aspects (by modifying and deleting elements – states, transitions, actions, etc.), while our framework offers only limited possibilities for this: for example, using the dynamic priority rules of Omega UML [6], one can give a higher priority to an aspect than to (a part of) the extended system, so that the extended system is blocked while the aspect executes, or use standard mechanisms in the aspect to terminate the execution of another object/slice.

Another framework that is somewhat related to ours is that of composition filters with superimposition [2]. Message filters are handlers which may intercept messages addressed to an object and perform some user-defined computation which can end by calling the corresponding method, by forwarding the message another object, or by taking any other course of action. However, in [2] filters only intercept messages that were explicitly sent to an object, and therefore there is no equivalent to the main feature of our proposal, which is to allow implicit communication between objects through observation.

6 Conclusions

This paper presents a modelling framework in which event observation is used as first-class object interaction mechanism. This is achieved mainly by formalizing the notion of *event* and its characteristics, and by defining a new modelling construct, the *probe*, which is used to match events and to manipulate event data.

The main (and motivating) application is the specification of tightly coupled components, such as those appearing in design methods based on use-case slicing or on view-based slicing. Frequently, the coupling between two components is unidirectional (*observed-observer*) and in such cases our framework reduces the burden of the designer by making part of this coupling implicit, and by avoiding most of the impact on the specification of the *observed* component.

We prototyped the constructs presented in this paper in a tool which translates (a subset of) VxUML to IF specifications [3]. This task was eased by the fact that IF is mainly dedicated to simulation and formal property verification, and it is easy to add a centralized event monitor to the platform. If the target is the final implementation, event observation must be done more efficiently, e.g. by code instrumentation based on a publish-subscribe pattern. We are currently experimenting with such an implementation. Among the difficulties we encounter, we note that the probe concept (in particular, access to event metadata) requires reflective support in the language in order to be fully satisfactory. Therefore, we orient our prototyping efforts on languages which offers such support (Java, Python).

References

[1] Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L.J., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In: Proceedings of OOPSLA. ACM, New York (2005)

[2] Bergmans, L., Aksit, M.: Composing crosscutting concerns using composition filters. Commun. ACM 44(10), 51–57 (2001)
[3] Bozga, M., Graf, S., Ober, Il., Ober, Iu., Sifakis, J.: The IF toolset. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 237–267. Springer, Heidelberg (2004)
[4] Cottenier, T., van den Berg, A., Elrad, T.: Stateful aspects: The case for aspect-oriented modeling. In: Workshop on AOM at the 6th International Conference on AOSD, Vancouver (2007)
[5] Douence, R., Le Botlan, D., Noyé, J., Südholt, M.: Concurrent aspects. In: GPCE Proceedings, pp. 79–88. ACM, New York (2006)
[6] Graf, S., Ober, Il., Ober, Iu.: A real-time profile for UML. STTT 8(2), 113–127 (2006)
[7] Harel, D., Kugler, H.: The Rhapsody semantics of statecharts (or, on the executable core of the UML). In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.) INT 2004. LNCS, vol. 3147, pp. 325–354. Springer, Heidelberg (2004)
[8] Hoare, C.A.R.: Monitors: An operating system structuring concept. Commun. ACM 17(10), 549–557 (1974)
[9] Jacobson, I., Ng, P.-W.: Aspect-Oriented Software Development with Use Cases. Object Technology Series. Addison-Wesley, Reading (2005)
[10] Jard, C., Monin, J.-F., Groz, R.: Development of Véda, a prototyping tool for distributed algorithms. IEEE Trans. Software Eng. 14(3), 339–352 (1988)
[11] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
[12] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
[13] Lakhnech, Y., Bozga, M.: IF-2.0 common language operational semantics. Technical report, VERIMAG (September 2002)
[14] McDougall, R., Mauro, J., Gregg, B.: Solaris(TM) Performance and Tools: DTrace and MDB Techniques for Solaris 10. Prentice Hall, Englewood Cliffs (2006)
[15] Nassar, M., Coulette, B., Crégut, X., Ebersold, S., Kriouile, A.: Towards a view based unified modeling language. In: ICEIS (3), pp. 257–265 (2003)
[16] Navarro, L.D.B., Südholt, M., Vanderperren, W., De Fraine, B., Suvée, D.: Explicitly distributed AOP using AWED. In: AOSD, pp. 51–62. ACM, New York (2006)
[17] Ober, Iu., Graf, S., Ober, Il.: Validating timed UML models by simulation and verification. STTT 8(2), 128–145 (2006)
[18] Object Management Group. Unified Modeling Language, `http://www.omg.org/spec/UML/`
[19] Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program 60-61, 17–139 (2004)
[20] Walker, R.J., Viggers, K.: Implementing protocols via declarative event patterns. In: SIGSOFT FSE Proceedings, pp. 159–169. ACM, New York (2004)
[21] Whittle, J., Moreira, A., Araújo, J., Jayaraman, P.K., Elkhodary, A.M., Rabbi, R.: An expressive aspect composition language for UML state diagrams. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 514–528. Springer, Heidelberg (2007)