

# Event Probes: A new Mechanism to support View-based Behavioral Modeling

Younes Lakhrissi<sup>2,3</sup>, Iulian Ober<sup>2</sup>, Mahmoud Nassar<sup>1</sup>, Bernard Coulette<sup>2</sup>, Abdelaziz Kriouile<sup>1</sup>

<sup>1</sup> Université Mohamed V-Suissi  
SI2M-ENSIAS, Madinat Al Irfane,  
BP 713, Rabat, Maroc  
{nassar, kriouile}@ensias.com}

<sup>2</sup> Université de Toulouse  
IRIT, UT2, 5 allées A. Machado, 31058  
Toulouse, France  
{iulian.ober, bernard.coulette}@irit.fr

<sup>3</sup> Université Mohamed V-Agdal  
ACSYS, Faculté des Sciences,  
BP 1014 Rabat, Maroc  
younes.lakhrissi@gmail.com

**Abstract** — Model composition is seen as a fundamental activity in model driven software development. Composition is a central problem when adopting a *multi-modeling* approach to analyze and design software systems. The term *multi-modeling* designates the approaches that involve the construction of several independent models for some entity of a software system, in order to separate different concerns. Depending on the rules and mechanisms used for separating concerns, multi-modeling approaches can be subject-driven [Tarr et al., 1999], aspect-driven [Jacobson et al., 2005], viewpoint-driven [Mili et al., 2001] etc. The work presented in this paper concerns view-based modeling, as it has been described in [Nassar et al., 2003, Anwar et al., 2010]. It is a variant of the object-oriented modeling approach for the analysis and design of complex systems, focusing on the actors that use the system and decomposing the specification of each class of the model according to the actors' needs. The proposal is made in the context of VUML, a language which supports view-based modeling. However the work achieved on the VUML profile does not tackle behavior modeling so far. In a previous paper [Ober Iu et al, 2008a, 2008b], we introduced the concept of event probe as a powerful mechanism addressing this question. In this article we propose a refinement of the introduced concepts and organize them in the form of a UML profile called *VUML Probe\_profile*.

**Keywords**—*View based modeling, VUML profile, VUML Probe\_profile, event observation, multi-view states machine, behavior composition.*

## I. 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 among 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.

The main application of this work is *view-based modeling*, a variant of the object oriented modeling approach for the analysis and design of complex systems, focusing on the actors that use the system and decomposing the specification according to their needs. With this prospect, our team developed a UML profile named VUML (View based UML), which allows the elaboration of a unique and sharable model accessible according to the view of each of the system's actors [Nassar et al., 2003, Nassar 2005]. The main new artifact of VUML is the multi-view class, formed of a *base class* describing the structural and behavioral features shared by all actors of the system, and of a set of *views*, each describing the features that pertain to a specific actor. Each view is linked to its base through a new relation stereotyped « view\_extension ». Dependencies among views are expressed through a new relation stereotyped « view\_dependency » and constraints expressed in natural language or in OCL. A *multi-view object* is an instance of a *multi-view class*.

However, previously to this contribution, the work achieved on the VUML profile did not tackle the modeling of behavioral aspects. The VUML approach addresses the structural aspects related to the composition of views and to the sharing of data without dealing with the behavior of views, i.e. how they react to invocations, and how they synchronize to achieve the behavior of multi-view objects.

In previous work our team treated the question of the composition of structural models [Anwar et al., 2010]. This question concerning models composition becomes more delicate when it treats composition of behavioral models. In our work [Ober Iu. et al, 2008a, 2008b], we introduced event probes as a fundamental concept addressing this problem. The subject of this work was the behavioral specification and composition of such object slices. The problem is known to be closer to aspect composition than to traditional interface-based composition. The new behavior specification construct is 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). We presented a modeling 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 modeling construct, the probe, which is used to match events and to manipulate event data.

The present paper concerns the refinement of the introduced concepts. We develop new mechanisms for better determining and identify precisely the concept of probe, such as, projection, the derivation and the composition of probes. We organize our proposal in the form of a profile called *VUML Probe\_profile*.

The paper is structured as follows: Section 2 presents the context of this work - view-based modeling - which had been concretized by the VUML profile, and its application on a case study. Section 3 describes the central concepts of our proposal: probes and related concepts. Section 4 presents the VUML Probe\_Profile whereas in section 5 we apply the profile on a case study. We end the paper by comparing our proposal to some existing approaches (Section 6), and by drawing conclusions and the main lines of future work.

## II. APPLICATION CONTEXT: VIEW BASED MODELING

The VUML profile [Nassar et al., 2003] was developed to meet the needs of complex information system modeling with UML according to various viewpoints. A viewpoint on the system represents an actor's requirements and rights. Such viewpoints may be considered as functional user-centered aspects. A view is the result of the application of a viewpoint to an entity (class), and by generalization to the entire system. The main concept of VUML language is the multiview class, which is composed of a base class (shared by all viewpoints), and a set of view classes (extensions of the base class), each view class being specific of a given viewpoint. Such a multiview class allows a fine-grained management of access rights and separation of concerns at the class level. VUML's semantics is described by a meta-model, including constraint rules expressed in OCL [Nassar, 2005].

### A. VUML analysis/design process

VUML aims to reduce the design complexity of software systems through decomposition according to the needs and access rights of the system actors. This horizontal separation of concerns completes the vertical approach of MDA by proposing a methodology that permits to develop models at each level of abstraction [Anwar et al., 2010].

The design process of VUML is made of three main phases composed of steps (see Figure 1). The first phase is the identification of actors' needs. The main goal is to create a requirements model (UML use case diagram). The second phase of the process, decentralized, consists in developing separate PIM models, each one representing a viewpoint. The result of this phase is a set of UML models (class diagrams, state machines, sequence diagrams, etc.). The last phase is a composition operation; it consists in merging design models developed separately in order to obtain a global VUML design model.

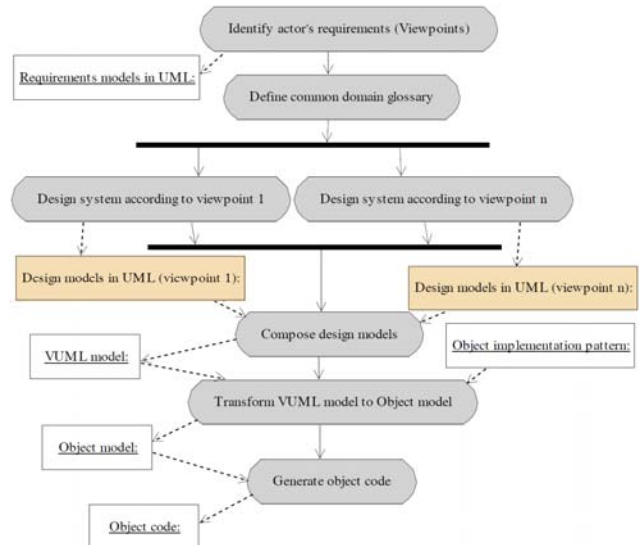


Figure 1. General VUML process [Anwar et al., 2010]

### B. Case Study

To illustrate our approach, we consider as case study (mainly developed in French, as one can see on figure 2) the management of an auto repair shop. It is a complex information system, involving several actors: manager, technicians, clients, etc. This case study details the construction of a system according to the VUML approach described above. We focus the study on the specification of the behavior of objects in the second phase and also on the composition of these behaviors in the phase of fusion (third phase).

To simplify, we limit our study to the following actors and activities:

- The client, owner of the car (*Client*): the system must allow the client to consult information concerning his car, to follow its evolution in the repair chain, to consult the repair and expert reports.
- The agency manager (*Manager*): the system must allow the manager to edit contracts with clients, to edit contracts with suppliers, to transmit orders to the employees to start the expertise and repair, and to ensure financial management.
- The mechanic (*Mechanic*): the system must allow the mechanic to consult the history of the failures, to edit and record the expert and repair reports, and to record the repaired parts.

The result of the composition of separate structural models is a VUML model that contains multiview classes resulting from the merging of homonymous classes defined separately. Figure 2 shows a subset of a VUML class diagram illustrating the multiview class *Car*. It contains a base - stereotype «base» - which is the shared part accessible by all the actors, and views - stereotype «view» - representing parts specific of each actor (Client, Manager, Mechanic).

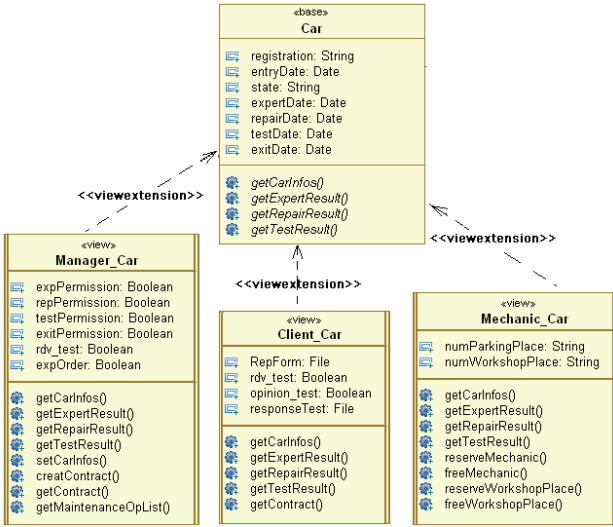


Figure 2. Excerpt of VUML class diagram: the multiview class Car

### III. PROBES

A probe is a modeling construct which serves to identify and manipulate events that are relevant for a particular goal. In particular, we use probes for modeling implicit interaction between objects. The aims of the probe event concept cannot be concretized using the existing modeling elements, especially in UML. In this section we present the necessary concepts that allow understanding our proposal.

We have identified groups of probe's that are matched with the usual events categories that occur during execution. By the concept of projection, we can refine these elementary types of probes to specify a precise behavior. We define the mechanism of probe derivation which permits to create new probe classes with additional attributes, but which always observes the same event type than the parent class. We provide another more complex concept to personalize a class of probes thanks to the composition mechanism. Composite classes are able to observe several types of events at the same time.

#### A. Elementary probe types: the Probe Library

After studying the types of events that can be produced during the system execution, we have identified three families of elementary probes (Figure 3). These elementary probe types are situated at the M1 modeling level according to the MDA terminology [Soley, 2000]. We define them in a *ProbeLibrary*. These elementary class probes give to the system designer the possibility to use them in the models as predefined classes. Each library type can be instantiated to produce concrete probes. As mentioned above, these types can be the subject of a context personalization by the means of mechanisms of projection, derivation and composition.

By their particular nature, the probes can handle data of model's level (M1 level) or meta-model's level (M2 level). For example, the *ObjectProbe* type handles the

*observedObject* type which is an object of M1 level, and the *class* type indicates its membership class, which is an element of the M2 level. To be able to do that, it is necessary to use a language that supports reflexivity.

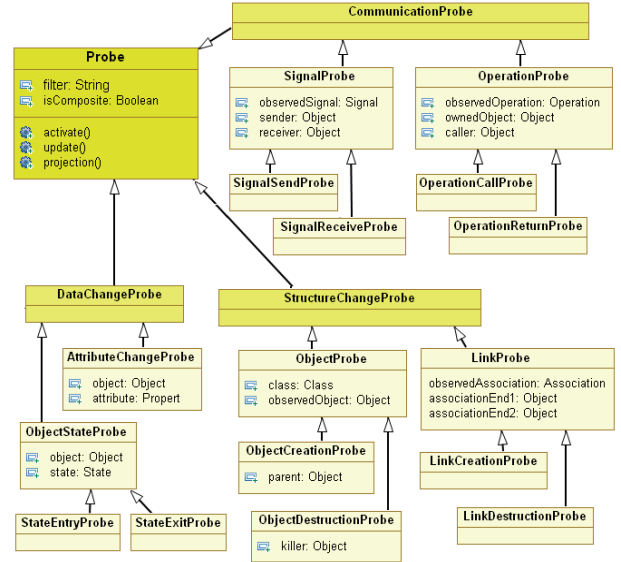


Figure 3. Elementary probe types: ProbeLibrary

#### B. Declaration and instantiation of probes

Like classical classes the probes declarations are done independently from the system entities. Once the structure of a probe class defined, the developer can instantiate it and use it in its design models. We illustrate on figure 4-(a) an example of instantiation of the *SignalSendProbe* class. The probe *repairOkObs* is instantiated in the model to detect the signal transmissions. If any constraint is listed in the *filter* attribute, the probe will become active for any signal transmission in the system; if not, the probe will select the events which respond to the constraint filter (see the following section about projection).

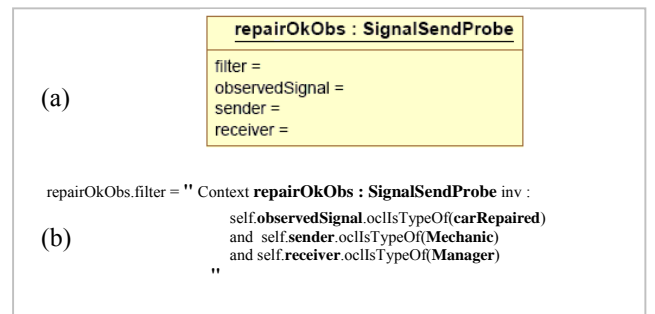


Figure 4. (a) example of probe instantiation (b) filter example

#### C. How to use probes in UML

In this section, we present how to use the probe concept

in a UML development through the following example. Let us consider *My\_class* as a class wishing to use the probe concept to detect the sending of the *startRepair* signals by the agency manager. To implement the observation, we follow the steps above:

1. Instantiate the probe class matched to the observed event type, which is, in our example, the *SignalSendProbe* type. We name the result instance *startRepObs*. This is done by declaring an *InstanceSpecification* UML element and then by stereotyping it as « probe ».

2. Specify the probe filter by describing the constraints to check at observation moment. In our example, we have two constraints to be applied to *observedSignal* and *sender* attributes.

3. Declare a reference to the instantiated probe. This is done by declaring a UML Reception inside the class and then by stereotyping it as « probeUse ».

4. Use the probe instance in the behavior specification within the states machine matched to the class.

Figure 5 gives an overview of the probe *startRepObs* used to describe the behavior of *My\_Class*. Figure 5-top illustrates the definition of the probe *startRepObs* and Figure 5-low shows the reference and the use of this probe by the states machine associated with the class *My\_class*.

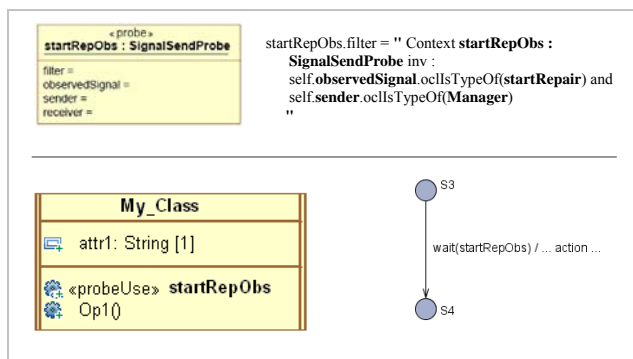


Figure 5. How to use probes in UML development

#### D. Probes Projection

Each type of probe is associated with a type of event and consequently the probe is activated for any appearance of this type of event in the system. This property is not always intended, because sometimes we need to filter the events on the context information basis. This requires the application of additional constraints which must be satisfied at the probe activation times.

The projection operation allows to apply additional constraints to the conditions of probe activation. It defines the context of the observation by adding conditions to be satisfied by the events matched to the probe type. This is achieved by the attribute *filter* that allows the expression, in a string character, of Boolean data and meta-data conditions of the observed events. We use the OCL language to express these constraints.

Figure 4-(b) gives an outline of the *SignalSendProbe* probe projection. The filter expressed here is a conjunction

of three OCL constraints to be checked by the events with the signal transmission type. The first constraint checks that the observed signal must be a *carRepaired* type; the second specifies that the transmitting object is a *Mecanicien* type; and the third constraint checks that the receiving object has as type *Manager*.

The probe is activated by an event only if the parameters of the event satisfy the filter conditions. The activation means that the set of attributes of the probe are updated with the parameters of the event and, in the state machines of objects using the probe, transitions waiting the activation of the probe become executable. This is done implicitly by the execution of the *activate()* operation.

#### E. Probes Derivation

The predefined probe classes of the ProbeLibrary contain the meta-data of the observed type event. The developer can declare new attributes to store additional information concerning the system's state and the moment of probe activation. This is allowed by deriving the type of the considered probe.

Let us consider a probe with *SignalSendProbe* type; this probe will be used for the detection of the *testOK* signal which is sent by the *Car* object type. Let us consider also, an instance of the *SignalSendProbe* probe with an adequate filter to carry out this observation. If we need - at the probe activation moment - to save the instant of the signal sending, we have to extend this probe by an additional attribute, *date* for example. Figure 5 explains the class of the derived probe '*SignalSendProbe\_testFonction*' (Figure 5-a) and an instance of this class with its filter (Figure 5-b).

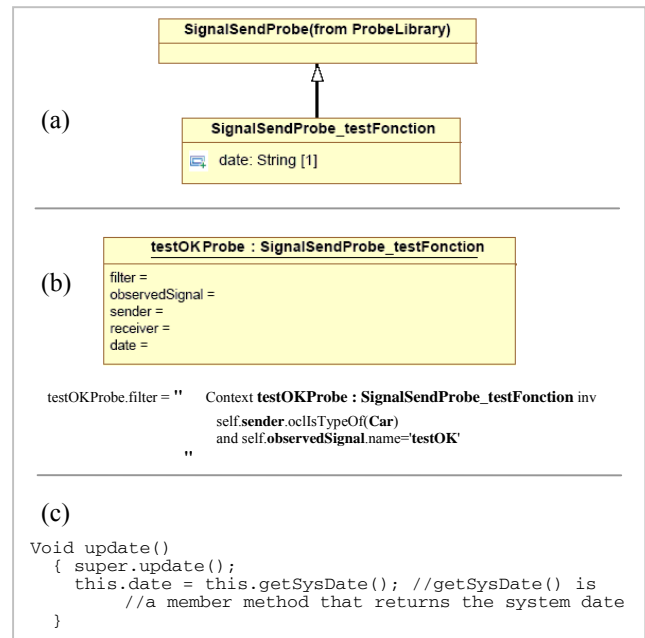


Figure 6. Probe derivation: example of testOKProbe

We can notice that the probe *update()* operation is responsible for the update of the probe attributes at activation moment. Consequently, this operation must be redefined in the derived class to be able to update the new added attributes (*date* in our example). Figure 5-c shows a code written in Java, that gives an overview of the *update()* operation of the class *SignalSendProbe\_testFonction*.

#### F. Composition of elementary probes

Until now, we have discussed two mechanisms able to customize the elementary probes. First, the projection mechanism, based on a filter, which provides the ability to customize a probe to a particular context using the data and meta-data events (section III.D). Second, the derivation of a probe, which offers the possibility to add new attributes in order to store additional information on the state of the system during the activation of the probe (section III.E). However, these two mechanisms can customize only one type of probe at once. Both mechanisms do not represent complex probes based on more than one type of event. In this section we present a third mechanism, the composition of probes, whose objective is to observe several types of events at the same time.

We have chosen to represent the composition of probes by a states machine. This choice is motivated by several reasons, among which are: (1) simplify the work of designers to express the forms of composition, i.e. express the composition in graphical form and not in mathematical formulas; (2) the opportunity to represent the timing appearance order of events in the system, (3) in general, benefit from the resources and facilities offered by a states machine concerning the description of behaviors in a state/transition form.

Consider the simple situation where we want to reference the moments of appearance of (*A* behavior followed by *B* behavior). Assume that *A* is the behavior observed by the probe *obs1*, and *B* is the behavior observed by the probe *obs2*. Figure 7-a illustrates the composition of the two probes *obs1* and *obs2* as a states machine.

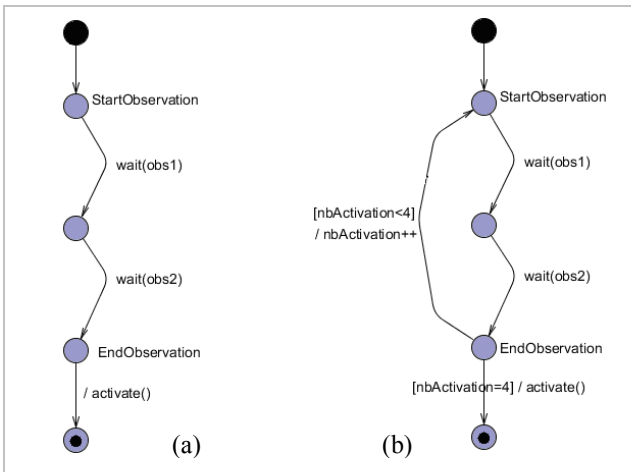


Figure 7. Examples of probe composition

On the same example, suppose that we want the probe to be launched after four sequences (*A* behavior, *B* behavior), the states machine representing the composition is given in Figure 7-b, where the variable *nbActivation*, initialized to 1, counts the number of the sequence realization. Note that between the two states *startObservation* and *EndObservation*, we can place any scenario associated with the definition of the composed probe. On leaving the state *EndObservation*, the operation *activate()* is executed, thus declaring the completion of the observed behavior.

#### IV. VUML PROBE PROFILE

The VUML Probe\_Profile extends the VUML metamodel by introducing a number of stereotypes: ProbeClass, Probe, ProbeUse, ProbeEvent, and Wait. These stereotypes are the result of the operation of mapping from the meta-model presented in details in [Lakhrissi, 2010]. Figure 8 provides an overview of the abstract syntax associated with the proposed profile.

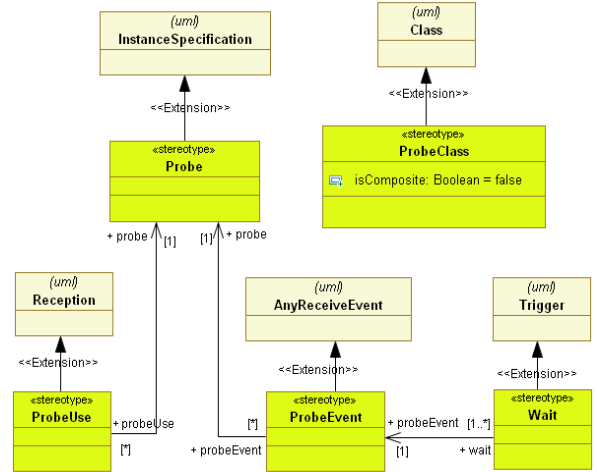


Figure 8. Probe\_Profile stereotypes definition

Elements of the meta-model are aiming to establish a correspondence between the UML concepts and domain concepts represented by our profile. In this section we present our choices for the translation into stereotypes of the proposed meta-model elements:

- The stereotype «probeClass» extends the UML *Class* metaclass. It is used to represent the elements of the library probe types. It also helps to distinguish between two types of probes (elementary and composite) through the tagged value *isComposite*. The value *false* (assigned by default to new created elements) denotes that the probes class is elementary.
- The stereotype «probe» extends the UML *InstanceSpecification* metaclass. It is used to denote objects that are instances of classes stereotyped by «probeClass». To ensure that, we have developed OCL constraints and we have associated them to the profile (see next section).

- The stereotype «probeUse» extends the UML *Reception* metaclass. A *Reception* in UML is a modeling element that announces the use of a signal by the class that declared the *Reception*. We have extended this meta-class because it is close to the semantics of *ProbeUse*. Syntactically, we denote a reference to a probe *obs1* as follows: «probeUse» *obs1*.
- The stereotype «probeEvent» extends the UML *AnyReceiveEvent* metaclass.
- The stereotype «wait» extends the UML *Trigger* metaclass. The syntax *wait(obs)* used in a state-machine transition specifies that the trigger *Wait* is waiting for the activation of the probe *obs* to cross this transition.

#### A. Well formedness Rules

We give here some semantic rules associated with the elements defined in *Probe\_profile*. They are described first in natural language and then translated in OCL language.

- All *InstanceSpecifications* stereotyped by «probe» have necessarily a classifier stereotyped by «probeClass».

```
Context InstanceSpecification def :
ProbeCondition1 : Boolean =
  if thisModule.inElements->includes(self) and
    self.hasStereotype('probe')
  then
    if not self.classifier.ocllsUndefined()
    then self.classifier->first().hasStereotype('probeClass')
    else true
    endif
  else true
  endif;
```

- A trigger stereotyped by «wait» cannot refer to an event.

```
Context Trigger def : ProbeCondition2 : Boolean =
  if thisModule.inElements->includes(self) and
    self.hasStereotype('wait')
  then
    if self.event.ocllsUndefined()
    then false
    else true
    endif
  else true
  endif;
```

- All elements having as type *Reception* and stereotyped by «probeUse» can refer only elements as type *Probe*.

```
Context Reception def : ProbeCondition2 : Boolean =
  if thisModule.inElements->includes(self) and
    self.hasStereotype('probeUse')
  then
    if self.signal.ocllsUndefined()
    then false
    else true
    endif
  else true
  endif;
```

## V. APPLICATION

One of the basic principles of multiview design approach is to ensure independence of the model-view development. However, this rule may be broken if there is a strong correlation among the models-view behaviors. To overcome this problem we have proposed the observation concept based on event probes.

For example, in the development of the client actor's states-machine, we need to know the real moment of the beginning of the car repair. But this information is not available in the client model-view, and we do not know as well who is the actor in charge of the starting of the repair (is that a decision of the agency manager? mechanic? workshop manager?). Finally we don't know how this action will be represented (in the form of a signal exchange? a message? an attribute change?).

The following sections present the solution principle using probes to define the views behavior independently of each other, and the behavior composition principle in the composition phase (see section II.A).

#### A. Abstract declaration of probes

Once the class diagram – related to a given viewpoint – is made, we develop the states machines corresponding to the reactive classes present in the view-model.

For example, if we consider the car states machine corresponding to the client-view, we need to know the beginning time of the car repair operation. Until now the reference event is not determined (it depends on another viewpoint which is not necessarily modeled) and its determination will be possible only in the viewpoints composition phase. Therefore, we declare an abstract probe named *repairStartProbe*. Figure 9-a shows the abstract declaration of the probe as a direct instance of the *Probe* class (root class in the *ProbeLibrary*).

#### B. Using probes in view models

The declared probe can be used in the behavior specification of the client viewpoint, precisely in the behavior specification of the *Car* class. To use it, we have to specify the probe reception inside the *Car\_Client\_View* class through the stereotype «probeUse». Then, the probe can be used in the behavior specification associated with this class through its states machine. In Figure 9-b, the probe *repairStartProbe* is used to trigger the transition between the states *ContractApproved* and *InRepair*.

#### C. Definition of abstract probes in the composition phase

The behavior composition, achieved after the structural composition, aims to describe the behavior of multiview reactive objects [Lakhrissi 2010]. A multiview object is composed of a set of objects, each one expressing a view behavior described by a machine developed in the view modeling phase. The mapping and synchronization between view machines is done through the concretization of the abstract probes, i.e. the definition of abstract parameters.

Figure 9-c illustrates the result of the composition step of models developed during the decentralized phase. Figure figure 9-c-top shows an excerpt of the Car class after structural merging with only one actor, the client. A car is well represented by the basis class *Car\_Base* shared by all viewpoints and by a set of views, each describing the features that pertain to a specific actor. Figure 9-c-low present the realization of the probe *repairStartProbe* after the parameters concretization previously undefined. The type of *repairStartProbe* is *SignalSendProbe*, and the observed signal name is *repairOrder*.

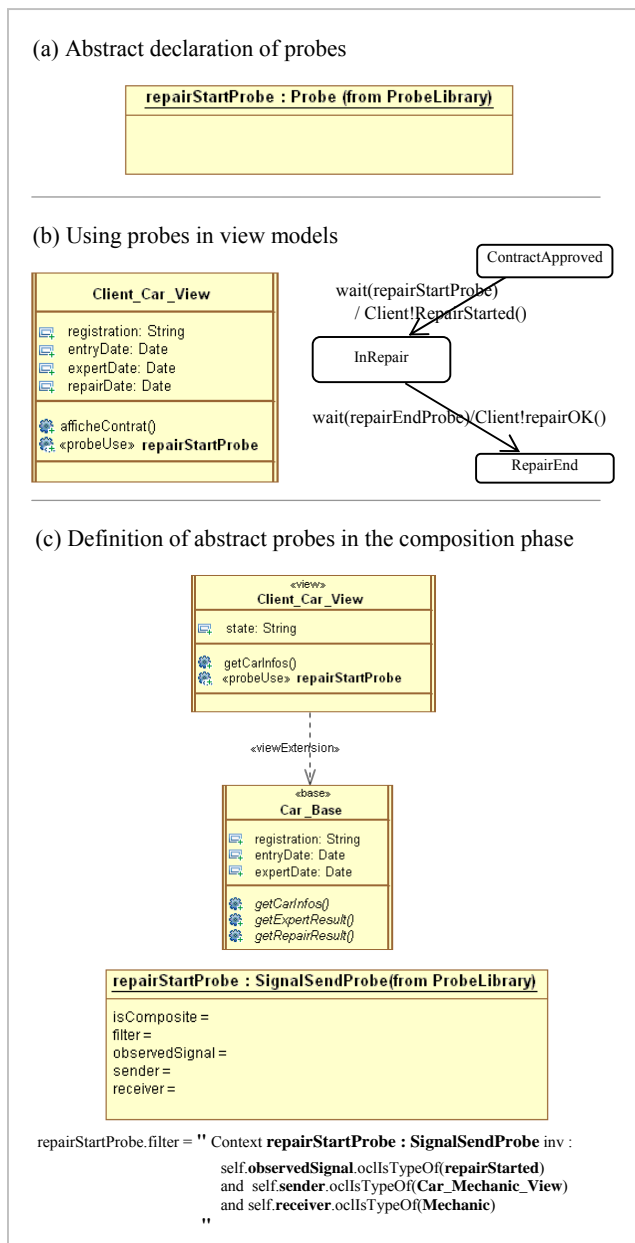


Figure 9. Using probes in VUML process

## VI. RELATED WORK

The work presented in this paper is a follow-up of [Oberlu et al, 2008a, 2008b]. There, we introduced the concept of probe, we defined its use for behavior specification and communication between object slices and we illustrated its application on small examples. In addition, the present paper describes how probes are integrated in the VUML profile and introduces more elaborate mechanisms for refining and composing event probes.

Event observation has previously been used in particular application settings such as formal verification, software testing and debugging, and some forms of aspect-driven modeling. However, to the best of our knowledge, event observation has never been studied in the context of object-oriented modeling languages and there are no other proposals of modeling languages integrating probes, event types or similar concepts.

**Observers in formal verification.** Event observation has been described as a mechanism for formal specification of properties in the Veda verification tool [Jard et al., 1988]. This concept has later been used in other verification tools for asynchronous concurrent systems such as [Algayres et al., 1995, Bozga et al., 2004]. Our approach was originally inspired by this work, and bears some similarities with it, like the existence of event types and parameters. However, the integration of our proposal in a powerful modeling language like UML allows us to go further, by modeling the event types, by including meta-parameters, refinement and composition mechanisms, etc. The use of event observation as a first class interaction mechanism between objects is also novel.

Event observation has also been proposed for the verification of synchronous models [Halbwachs et al., 1993]. However, in synchronous models, the only event types are data changes, which can already be specified in synchronous languages, and therefore there is no need for any specific language mechanism for modeling observation. The computational expressiveness of (distributed) event observation has also been extensively studied in decentralized supervisory control of discrete-event systems (e.g., [Lin et Wonham, 1988]), but without linking it to any concrete language for specifying observation.

**Event observation in aspect-oriented programming.** Several approaches in aspect oriented programming propose the specification of joint points using (sequences of) events [Walker et Viggers, 2004, Allan et al., 2005, Navarro et al., 2006, Douence et al., 2006]. The computational model that is closest to ours is the Concurrent Event-Based AOP (CEAOP) defined in [Douence et al., 2006], 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 [Douence et al., 2006] events are just simple (uninterpreted) synchronization labels.

**Other uses of event observation.** Some recent efforts in the area of distributed event-based systems (DEBS) aim also at the definition of a specific language for event detection

(e.g., [Cugola et Margara, 2010]). The main difference with our approach is that the language of [Cugola et Margara, 2010] is textual and not integrated in a general-purpose modeling language like UML. The application domain, which is that of large scale event stream processing, is also quite different from ours, leading to different language choices.

Other forms of event observation may be found in the area of program tracing and debugging (e.g., technology D-Trace of Sun [McDougall et al., 2006]) or in the field of autonomous agents [Kaminka et al., 2004].

## VII. CONCLUSION AND FUTURE WORK DIRECTIONS

This paper presents a UML profile, which extends the VUML profile for view-based modeling, and which supports event observation as first-class object interaction mechanism. Our profile formalizes the notion of event probes, identifies the different event probe types relevant in the scope of VUML, and defines mechanisms for event probe refinement and composition.

The underlying motivation for introducing these constructs is our search for a suitable behavior specification and composition mechanisms for views, given that view-based modeling inherently leads to very tightly coupled specifications. With traditional object interaction mechanisms (message passing), such specifications are cluttered by the large number of interactions necessary in order to synchronize views and make them cooperate. Event observation helps in that it can render a part of these interactions implicit. The use of the new language constructs is demonstrated in the paper based on an example, the multi-view modeling of a car repair agency.

**Future work.** Several directions for future work remain open. They concern the language, the supporting methodology and the tools. On the language side, the use of more elaborate mechanisms of UML, such as *templates*, could be done in order to facilitate the separate specification of view-points depending on abstract probes, which would become template parameters. View integration would then be supported by template instantiation with concrete probes.

As we previously mentioned, view-based modeling bears some similarity with *aspect-oriented modeling* (AOM). The solution proposed in this paper is however not fully adapted for AOM, since the specification of aspects sometime needs to be intrusive (i.e., restrict or change the behavior of the base model on which an aspect is applied) – something that is not possible using only the mechanisms proposed here. A direction of future work is to develop the present proposal into a full-fledged AOM language.

Another work direction concerns the integration of high-level inter-object behavior specifications in VUML. For now, the profile only concentrates on intra-object behavior specification by methods and state machines.

Finally, on the tools side, the prototyping tool described in [Lakhrissi, 2010] is to be considered only as a proof-of-concept. The tool has to be developed into a full code-

generator, in order to show that our approach can scale up to realistic models. This tool, as well as the various other tools developed around VUML (such as [Anwar et al, 2010]), would also need to be integrated in a consolidated platform.

## REFERENCES

- [Algayres et al., 1995] Algayres, B., Lejeune, Y. et Hugonnet, F. (1995). GOAL : Observing SDL behaviors with GEODE. Dans Braek, R. et Sarma, A., éditeurs : SDL'95 with MSC in CASE. Elsevier Science B.V.
- [Allan et al., 2005] Allan, C., Avgustinov, P., Christensen, A. S., Hendren, L. J., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G. et Tibble, J. (2005). Adding trace matching with free variables to AspectJ. Dans Johnson, R. E. et Gabriel, R. P., éditeurs : Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA, pages 345–364. ACM.
- [Anwar et al, 2010] Anwar A., Ebersold S., Coulette B., Nassar M., Kriouile A.: A rule driven approach for composing Viewpoint-oriented Models. Journal of Object Technology, Vol. 9, n°2, March-April 2010.
- [Bozga et al., 2004] Bozga, M., Graf, S., Ober, I., Ober, I. et Sifakis, J. (2004). The IF Toolset. Dans Bernardo, M. et Corradini, F., éditeurs : Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures, volume 3185 de LNCS, pages 237–267. Springer.
- [Cugola et Margara, 2010] Cugola, G. et Margara, A. (2010). TESLA : a formally defined event specification language. Dans Bacon, J., Pietzuch, P. R., Sventek, J. et C. tintemel, U., éditeurs : Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, DEBS 2010, Cambridge, United Kingdom, July 12-15, 2010, pages 50–61. ACM.
- [Douence et al., 2006] Douence, R., Botlan, D. L., Noye, J. et Suddholt, M. (2006). Concurrent aspects. Dans Jarzabek, S., Schmidt, D. C. et Veldhuizen, T. L., éditeurs : Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings, pages 79–88. ACM.
- [Halbwachs et al., 1993] Halbwachs, N., Lagnier, F. et Raymond, P. (1993). Synchronous observers and the verification of reactive systems. Dans Nivat, M., Rattray, C., Rus, T. et Scollo, G., éditeurs : Algebraic Methodology and Software Technology (AMAST '93), Proceedings of the Third International Conference on Methodology and Software Technology, University of Twente, Enschede, The Netherlands, 21-25 June, 1993, Workshops in Computing, pages 83–96. Springer.
- [Jacobson et al., 2005] Jacobson, I., Ng, P.-W.: Aspect-Oriented Software Development with Use Cases. Object Technology Series. Addison-Wesley, Reading (2005)
- [Jard et al., 1988] Jard, C., Monin, J.-F. et Groz, R. (1988). Development of Veda, a Prototyping Tool for Distributed Algorithms. IEEE Trans. Software Eng., 14(3):339–352.
- [Kaminka et al., 2004] Kaminka, G. A., Gmytrasiewicz, P., Pynadath, D. et Bauer, M., éditeurs (2004). Modeling Other Agents from Observations (Workshop Proceedings).
- [Lakhrissi, 2010] Lakhrissi Y. Intégration de la modélisation comportementale dans la conception par points de vue. Thèse de doctorat, Université de Toulouse, July 2010.
- [Lin et Wonham, 1988] Lin, F. et Wonham, W. M. (1988). Decentralized supervisory control of discrete-event systems. Inf. Sci., 44(3):199–224.

- [McDougall et al., 2006] McDougall, R., Mauro, J. et Gregg, B. (2006). Solaris(TM) Performance and Tools : DTrace and MDB Techniques for Solaris 10 and OpenSolaris. Prentice Hall.
- [Mili et al., 2001] H. Mili, H. Mcheick, J. Dargham, S. Dalloul. "Distribution d'objets avec vues". Revue L'Objet-7/2001, LMO'2001, pp. 27-44, 2001.
- [Nassar et al., 2003] M. Nassar, B. Coulette, X. Cregut, S. Ebersold, and A. Kriouile. Towards a view based unified modeling language. In ICEIS (3), pages 257–265, 2003.
- [Nassar, 2005] Analyse/conception par objets et points de vue : le profil VUML. PHD thesis, Institut National Polytechnique de Toulouse, september 2005.
- [Navarro et al., 2006] Navarro, L. D. B., Suñer, M., Vanderperren, W., Fraine, B. D. et Suve'e, D. (2006). Explicitly distributed AOP using AWED. Dans Filman, R. E., editeur : Proceedings of the 5th International Conference on Aspect-Oriented Software Development, AOSD 2006, Bonn, Germany, March 20-24, 2006, pages 51–62. ACM.
- [Ober Iu et al., 2006] Ober, Iu., Graf, S., Ober, Il.: Validating timed UML models by simulation and verification. STTT 8(2), 128–145 (2006).
- [Ober Iu et al., 2008a] Ober Iu., Lakhrissi Y. Observation-based interaction and concurrent aspect-oriented programming (2008). In International Conference on Software Engineering Research, Management and Applications (SERA 2008), Prague, Rép. Tchèque, 20/08-22/08/2008, Roger Lee (Eds.), Springer, SCI 150, p. 141-156, august 2008.
- [Ober Iu et al., 2008b] Ober Iu., Coulette B., Lakhrissi Y. (2008) Behavioral Modelling and Composition of Object Slices Using Event Observation. In ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2008), Toulouse, 28/09/2008-03/10/2008, Jean-Michel Bruel, Krzysztof Czarnecki, Ileana Ober (Eds.), Springer, LNCS 5301, p. 219-233, september 2008.
- [Soley, 2000] Soley et al. MDA Model Driven Architecture. Richard Soley and the OMG Staff Strategy Group, Object Management Group White Paper, Draft 3.2 – Nov. 2000.
- [Tarr et al., 1999] P.L. Tarr, H. Ossher, W. Harrison, M. Stanley, Jr. Sutton. "N Degrees of Separation : Multi-Dimensional Separation of Concerns". International Conference on Software Engineering, pp. 107-119, 1999.
- [Walker et Viggers, 2004] Walker, R. J. et Viggers, K. (2004). Implementing protocols via declarative event patterns. Dans Taylor, R. N. et Dwyer, M. B., editeurs : Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2004, Newport Beach, CA, USA, October 31 - November 6, 2004, pages 159–169. ACM.