Proposition of the Probe-Event Approach for View-Based Modeling

CHAIMAE OUALI-ALAMI, ABDELALI EL BDOURI, YOUNES LAKHRISSI
SIGER Laboratory,
Sidi Mohamed Ben Abdellah University,
Fez,
MOROCCO

Abstract: - Viewpoint modeling is the general theme of our work in the field of Model Driven Engineering. It is an object-oriented modeling strategy that focuses on the actors interacting with the system in order to analyze and create complex systems.

Building complex computer systems remains a particularly challenging process for the modeling, design, and analysis team despite the progress of design approaches in the field of software engineering due to the complexity and richness of information. Complex software system modeling is an extremely sophisticated and enormous area of study. The best method for reducing complexity and dimension while simultaneously making it easier for people to design complicated systems is to break them down into smaller parts or components. Thus, the concept of multi-modeling methods, So the composition of the models of the findings then poses a challenge.

To achieve this goal, we introduced the notion of event probe, which allows specifying implicit communications between views by observing events. This makes it possible to decouple specifications that are a priori strongly interconnected, to design them separately by viewpoint, according to the recommendations of the view modeling approach, and then to integrate them without having to modify them. We first defined the concept of event probes, identified the different types of probes with their associated parameters, and then defined a set of concepts allowing enriching and manipulating the probes.

Key-Words: - MDE, Viewpoint modeling, View-UML, event probe, UML, Behavior specification, Composition.

Received: June 23, 2022. Revised: April 25, 2023. Accepted: May 24, 2023. Published: June 26, 2023.

1 Introduction

In spite of the advancement of plan methods within the zone of program designing, the development of complex computer frameworks remains a complicated errand, [1]. In this setting, it is regularly inconceivable to develop a worldwide demonstration that considers all needs at the same time. The application is actually divided up into a number of models, which reduces the estimated complexity of the system. In each scenario, a compositional phase is necessary to produce the application's final adaption.

Figure 1 depicts the structure of a multi-view class. The stereotypical data classes "base" and "view" exhibit the static nature of the system. On the other hand, the state machines ("machine-base" and "machine-view") connected to these classes represent the behavior, [2].

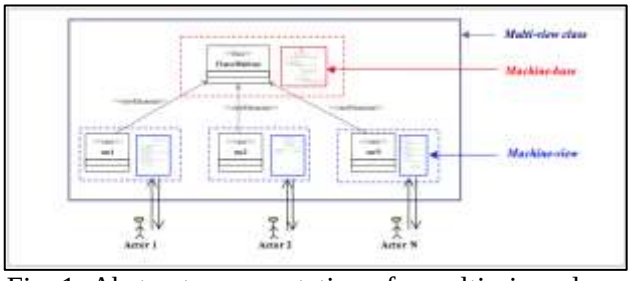

Fig. 1: Abstract representation of a multi-view class

A multi-view state is defined as the state that represents a multi-view object. It's a state with multiple meanings depending on the actors engaging with the system, and it's characterized by the following sub-states:

- The "machine-base" base-state: an abstract state that represents a point in a multi-view object's life cycle.
- The "machine-view" is made up of a series of view-states: states emerged from the refinement of the base state, taking into account the perspectives of system actors.

The objective of this paper is to answer the problem of behavioral specifications in the context of VUML profile, [3]. It consists in proposing new mechanisms that extend those of UML.

Our approach is based on an implicit communication between the views, through the observation of events. We point out that at the level of terminology, the notion of 'machine base' does not exist anymore in this approach. However, the term 'machine view' keeps its meaning and represents the state machine of a class of the system according to the considered view.

We propose a behavior specification technique, which we call an event probe, [4], based on the observation of the events of the system during execution. It addresses the problems related to the specification and composition of the behaviors of the different view models. Indeed, the behaviors of the view models evolve independently from each other and each of them uses and generates a set of events. Our proposal consists, in the view integration phase, in using the observation of events (and information related to these events) as a means of communication between the views.

This paper is composed of three principal parts. The first part explains the view modeling approach (Design by viewpoint) one of a system's decomposition procedures. The second part of our study defines the principal concept of observation based on the notion of event probe. The third part defines the basic concepts we have developed to handle probes, then we will explain how to declare and instantiate a probe, and we will explain the principle of projection as well as the derivation of the probes.

2 View Modeling Approach

2.1 Modeling Decomposition Principle

It takes a lot of analysis, modeling, and design work to implement technology widely throughout many fields, including computer science, mechanics, industry, economics, and commerce.

Conceptual models serve two purposes: (1) to understand a topic and its setting, and (2) to provide a framework for research and advancement. It is simpler to choose the right course of action for this task. As the number of users and the breadth of their requirements increase, so does the complexity of the subject under study. Solving such difficult systems has become essential. In order to manage and analyze a complex system, the system decomposition approach divides it into manageable problems and then chooses an acceptable solution, [1].

Quite apart from the progress of analysis and design methodologies in the field of software engineering, [31], whose construction of the global model remains a challenging work. It is required to determine the needs of the actors, despite the technological requirements, [5]. Multi-modeling approaches are model-oriented methodologies that employ independent model development. It is crucial to return to object-oriented approaches, [6], [7], [8], to analyze these so-called model-oriented approaches.

We are increasingly employing so-called multi-model modeling methodologies to deal with this complexity. This approach provides good decomposition practices.

Separation of concerns is critical when dealing with the complexity of large software systems since it keeps the event process, the resulting models, and hence the code manageable. The separation of concerns is often accomplished in a variety of methods, but the goal remains the same: the ability to recognize relatively distinct "pieces", [9], [10], [25].

We have focused in this article on the concept of point modeling, which is one of several ways and methodologies for the decomposition of complex systems, [1].

2.2 View-based Modeling

Viewpoint is defined in Robert's Dictionary as: "One must position oneself in a place in order to see an object as best as possible or as a particular way of looking at a problem". The terms closest to defining a viewpoint are appearance, optics, perspective, and view. In computer science, the concept of this point of view has several meanings that vary by job and field. Numerous disciplines of information processing, including databases, knowledge representation, analysis and design, programming languages, software engineering tools, etc., have examined viewpoints and views.

In the database world, the concept of a view is used by languages as a data selection function. When representing knowledge, views are used to describe categorical reasoning. A viewpoint specifies a set of characteristics related to a concept or family of objects. A concept can be viewed from different perspectives, [1].

To integrate the concept of perspective into the analysis/design of software systems. Nassar's work has led to the establishment of a UML profile called View Based UML (VUML), [28] that can analyze

and design software systems through an object-perspective approach, [11]. In the area of the database, view terms are used by the query language as a data selection function. In knowledge representation, views are used to justify taxonomic classifications and to represent knowledge representation. A view specifies a set of characteristics associated with a family of concepts or objects, [12].

The level of deconstruction in the view approach is different from the level anticipated in the aspect method. This is a breakdown from the perspective of the system's actors. Views are developed without making a distinction between fundamental functions and one another. This kind of deconstruction yields a collection of entities that are each subjectively described by one of the system's participants, [13], [14].

In order to solve the limitations of object-oriented programming, the VUML language (View-based Unified Modeling Language) is a UML profile based on the point-of-view modeling technique. Specifically, the implementation of views, [15], [16], [17], through the use of many instances. This approach focuses on the construction of various partial models while employing the idea of point of view to analyze and create a software system. It is most frequently used to define an information system that exhibits high user engagement and whose actor is either a person or another entity that interacts with the system, [18].

2.3 Methodology

Interactions embody the dependencies between services. In this context, you can identify two types of dependencies. Structural and behavioral dependencies, [19], [21], [29].

Structural dependency: From a structural point of view, if the s1 service requires the functionality provided for the operation of the s2 service when designing the s1 service, the s1 service depends on the s2 service. The s1 service is said to be structurally dependent on the service. Service s1 explicitly defines a dependency on service s2. Service s1 explicitly defines a dependency on service s2. Structural dependencies arise from structural interaction types. x as an example of a structural dependency, suppose the manager consists of an operator's service production report and a graphics service. Managers use Reporting Services to report on operator production stored on durable media. Operator service production reports use the display features provided by the graph service to provide manager user reports in a variety of graph formats. The operator's service production report depends on the graphics service.

Behavioral dependency: A s1 service is dependent on a s2 service when the implementation of the s1 service can influence the operation of the s2. The s1 service is structurally independent of the s2 service. Although s1 does not require the functionality offered by s2. A form of behavioral interaction manifests behavioral reliance. The Reporting Manager, for example, is linked to the Security Manager for the execution of service production reporting of operators, and graphic design is employed in the implementation of authentication: the data used for authentication corresponds to the data loaded. The Reporting Manager does not require the security manager's functionality, and the security manager does not require the Reporting Manager's functionality, but their performance is interdependent.

The work carried out on the VUML profile has so far focused on the structural aspect of modeling, but without taking into account the behavioral aspect of Multi-view modeling. In fact, the work carried out dealt with the static structuring of VUML applications, such as data sharing and static composition of views, without dealing with how these views will react, nor how to synchronize them to represent the behavior of Multi-view objects (Multi-view class instances).

Behavioral modeling is an important step in the design of a complex system [29], [30], especially in the context of model-driven engineering, where the objective is to automate the post-conception phases (coding, integration, validation, etc.), automation which must be based on the most complete design model possible.

UML behavioral modeling can be done at several levels of abstraction, starting from overall models such as interaction and activity models that represent the interactions and sequence of activities between different objects or components of the system, up to a fine description of the behavior of objects or components by state machines. Overall models, such as the sequence diagram, allow by definition the description of a behavior from one point of view or a combination of several points of view.

3 Principles

In this section, our team introduced the concept of event probe, which allows us to specify implicit communications between views by observing events. This makes it possible to decouple specifications that are a priori highly interconnected,

to conceive them separately by points of view, according to the recommendations of the VUML method, then to integrate them without having to modify them.

3.1 Event Probe Concept

A system run is considered an event trace. An event represents the smallest change in the state of that system. It represents the execution of an atomic action indicating the occurrence of a particular event. Events are of several types: calling an object method (synchronous communication), receiving a signal (asynchronous communication), changing the state of a Boolean condition, triggering a transition in a state machine, entering a state, etc.

We define the event probe term as a modeling element that identifies an event or sequence of events and uses it in the behavior specification during the design phase. This use can take several forms such as searching and detecting a given behavior in the system, controlling the same critical states, triggering behaviors following the realization of other behaviors, etc. As examples of probes, (i) probe to refer to all events of type "creation of an object of a given class "C", (ii) probe to detect and refer to any event of type "sending a signal of type "S" in the system", (iii) probe detecting the event "reception of the signal of type "S1" by an instance of class "C", etc. Generally speaking, a probe is defined in relation to a particular type of event (creation of an object, sending of signal, call of operation, etc.).

Each execution event occurs in a particular context. Probes provide access to information related to this context, which depends on the type of event. For example, for an event of type "object creation", the context information is the class of the created object, the identifier of the created object, the identifier of the "parent" object (who created it), etc. This data is stored as attributes of the probe.

At the modeling level, defining a probe amount to defining an instance of a predefined probe type (a library class). In addition to the type, the definition may specify a condition (or several) that an event must meet to activate the probe. This condition relates to the contextual information about the event (the probe attributes) and is implemented based on the probe projection technique presented in the rest of the article. Probe semantics specify that when the occurrence of an event activates a probe, the probe attributes are updated with the event contextual data. In summary, the purpose of the probes is to allow access to events during a run, that is to say, to access the data corresponding to these events (the identifier of the element triggering the event, the identifier of the target element of the event, the parameters transmitted, etc.) as well as their metadata (such as the class of objects concerned by the event, etc.). Once the probe of a given event is triggered in the system running, the probe attributes store the data and metadata related to that event.

3.2 Probe Operation

The ability to recognize events and utilize them to specify the behavior of objects is the purpose of the definition of a probe. This is accomplished with a design that enables an object to wait for the occurrence of an event that will be captured by a "obs" probe. We refer to this construct as wait(obs), and it is defined as a new kind of behavior trigger (Trigger) that may be applied to state machine transitions, [17]. Figure 2 below explains the principle of inter-object communication based on event probes.

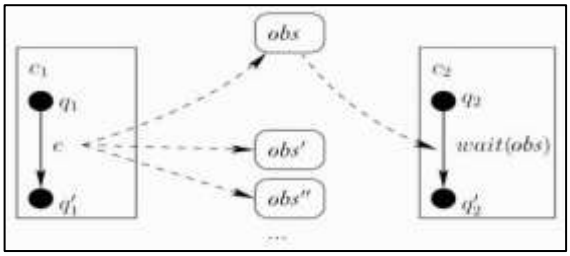

Fig. 2: Principle of communication-based on the probe concept

When a c1 object performs an action, it produces an «e» event that can be observed by one or more probes («obs», «obs'», «obs"» in Figure 2). Another c2 object can observe the event using a transition triggered by wait(obs). The communication between c1 and c2 is implicit insofar as in the developed model no explicit communication mechanism is specified by the user to inform c2 of the realization of the «e» event of the c1 object. The transition kept by "wait(obs)" can only be crossed if c2 is already in the q2 state when «e» occurred. c2 becomes executable immediately.

By making an analogy with the exceptions in object languages, we can make the parallel between the probe objects described here and the objects used to transmit predefined exceptions lifted for example by a Java virtual machine (for example: ArithmeticException, NullPointerException…). The activation of a probe and the updating of its attributes are done implicitly, as for the lifting of the predefined exceptions. However, the probe can then be used or not to trigger a behavior.

4 Definition of Concepts for Probe Handling

In our previous work, [1], we tried to specify the Multi-view behavior using only UML concepts, but the results of the approach show limitations (1) in relation to the behavior specification when we encountered difficulties to ensure independence in the development of the views because they can be strongly inter-coupled and (2) in relation to the behavior composition; The integration of the separately developed view-machines can require many modifications and adaptations at the level of the view-machines and the base-machine.

Consequently, we opted for another solution proposing new mechanisms specific to the modeling and the composition of the behaviors of Multi-view objects, hence the notion of event probe

The ability to recognize events and utilize them to specify the behavior of objects is the purpose of the definition of a probe. This is accomplished with a design that enables an object to wait for the occurrence of an event that will be captured by a "obs" probe. We refer to this construct as wait(obs), and it is defined as a new kind of behavior trigger (Trigger) that may be applied to state machine transitions, [17]. The probe derivation principle was then defined. It is a method that enables the creation of new probe classes that constantly observe the same kind of event as the parent class but with additional properties. We provide composition as a different method for altering a class of probes. In fact, the designer is able to develop new composite classes, or classes that can simultaneously observe many sorts of events.

4.1 Basic Probe Types: Probe Library

We have discovered three families of basic probes by examining the events that can be generated while a system is being used. (cf. Figure 3): (i) probes of communication events, such as sending/receiving signals and calling/returning operations; (ii) event probes related to changes in system structure, such as creating/destroying objects and creating/destroying links between objects; and (iii) data modification probes.

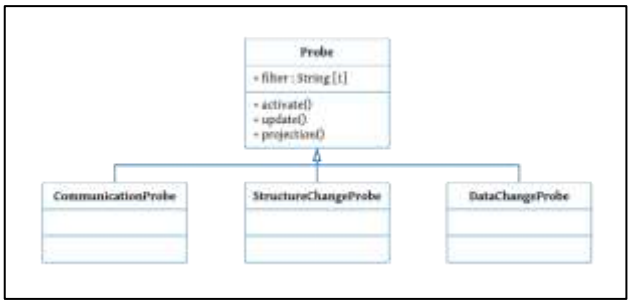

Fig. 3: Probe main types

According to MDA terminology, [8], these types of elemental probes are at the M1 modeling level. We've defined them in a ProbeLibrary library so that designers can utilize them as preset classes in design templates. To create actual probes, any type in this library can be created. These kinds, as previously indicated, are subject to content customization through projection, derivation, and composition.

Probes can handle data at the model level (level M1) or meta-model level (level M2) depending on their unique characteristics. A probe of type ObjectProbe, for instance, manipulates the type observedObject, which is an object at level M1, and the type class, which indicates the element's class of membership at level M2. As a result, in order to manipulate the probes, a language that enables reflexivity—that is, a language that can modify data at levels M1 and M2—is required. UML does not recommend this, but in principle, it is possible to use meta-model elements in a UML model. For implementation, we can cite Java as a language of level M1, which presents support, although limited, for level M2.

The following three sections provide definitions for the three basic types of probes proposed. It should be noted that at this stage it is difficult to give concrete examples of the use of the various probes. Detailed examples of these types of probes will be provided following the presentation of the projection operation.

4.2 Communication Probe

Communication probes (CommunicationProbe) allow the observation of events related to explicit interactions between objects. These probes can observe and reference signal exchanges between system entities. They can also observe and reference method calls and their returns between system objects (Figure 4).

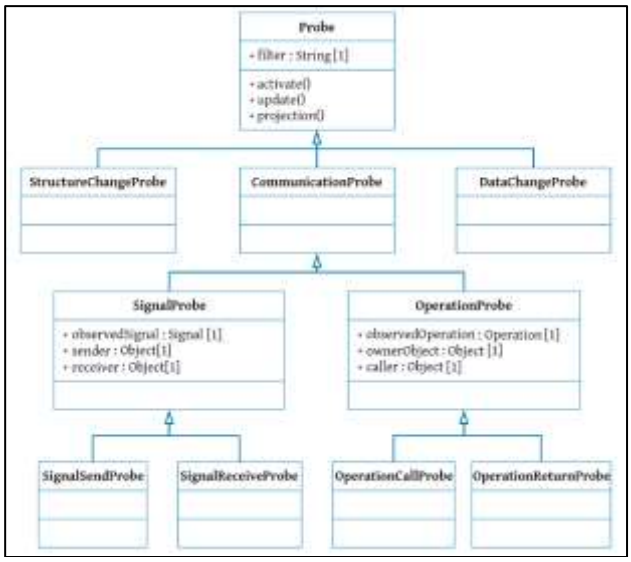

Fig. 4: Communication Probe

Signal Probes (SignalProbe)

- **SignalSendProbe:** This type of probe is used to reference signal emissions between running system objects.
- **SignalReceiveProbe:** This type of probe is used to reference signal reception by a running system object.

Operation Probes (OperationProbe)

- **OperationCallProbe**: This type of probe references operating calls between running system objects.
- **OperationReturnProbe:** This type of probe references returned from operations.

4.2 System Structural Change Probes

Structural change probes (StructureChangeProbe) allow the observation of events related to structural changes in the system. Thus, these probes may refer to the life of objects (creation/destruction of an object), but also to the structural relationships between entities, such as the creation and destruction of links (Figure 5).

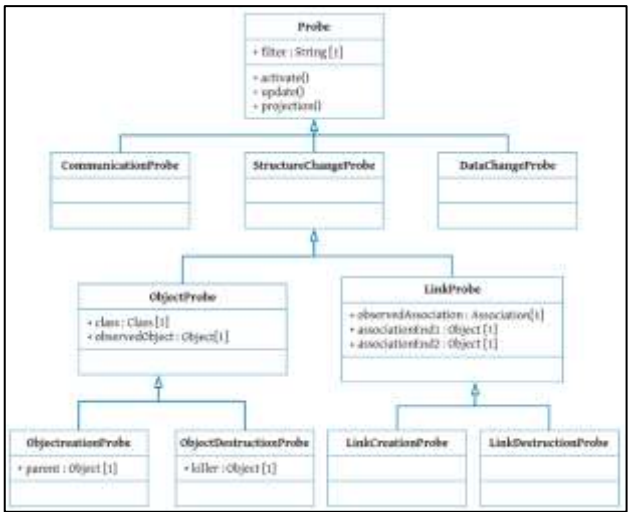

Fig. 5: Structure Change Probe

Object Life Probes (ObjectLifeProbe)

- **ObjectCreationProbe:** This type of probe is used to detect and reference creations of new objects in the running system.
- **ObjectDestructionProb:** This type of probe detects object destruction in the running system.

Once the probe is activated, it stores the variables of the context of the object concerned (in the first case the created object, and in the second case the destroyed object) for use by the object using this probe.

Example: to control the total number of objects in the system, we can declare two probes of type ObjectLifeProbe: the first (obs1) to reference all object creations, the second (obs2) to reference the destruction of objects in the system. The Figure 6 below shows the declaration of the two probes (Figure 6-a), as well as their use by the controller object (Figure 6-b). The nbObjects variable is used to store the number of objects in the system.

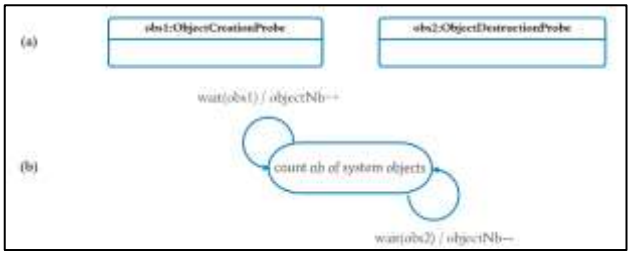

Fig. 6: Probes examples ObjectLifeProbe

Linkage Probes (LinkProbe)

- **LinkCreationProbe:** This type of probe detects the creation of new links between system objects.

- **LinkDestructionProbe:** This type of probe is used to reference link destruction between system objects.

4.3 Data Change Probes

Data Change Probes (DataChangeProbe) concern two types of events: events related to changes in the value of a system attribute, and events related to state changes in a state machine of an object (Figure 7).

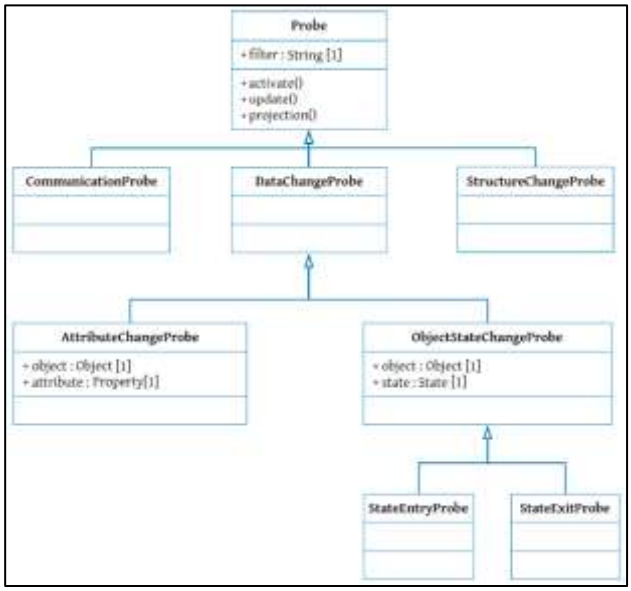

Fig. 7: Data Change Probe

Attribute Value Change Probes (AttributeChangeProbe)

This type of probe allows the observation of changes in attribute values of running system objects. The probe becomes active with each value change of an attribute of one of the system objects. This type of probe is dangerous to use directly in the system. If the probe activates for any change of attribute, this may cause the probe to be in an infinite loop, as the activation of the probe itself is done by attribute changing. The concept of projection, which we present in the next section, allows the probe to focus on an attribute or set of well-identified attributes.

Object State Change Probes (ObjectStateChangeProbe)

This type of probe is used to reference changes in the status of running system objects. A distinction is made between the entry detection probes in a new state and those for the state exit detection.

- **StateEntryProbe:** This type of probe is used to reference entries in new states of running system objects.
- **StateExitProbe:** This type of probe is used to reference state outputs of running system objects

4.4 Structure of The Predefined Class: Probe

A type of probe is a particular modeling element capable of storing and manipulating data and metadata. We define this element as a classifier likely to have attributes, operations, and state machines (to describe the behavior of the probe). We differentiate between two categories of probes: elementary probes and composite probes through the isComposite attribute. A predefined probe has a set of predefined attributes and operations that we present below:

4.4.1 Predefined Attributes

The attributes of a probe store data and metadata to be stored in relation to the event observed by the probe. The predefined attribute common to all probe types is:

- **filter**: an attribute that describes the Boolean conditions on the event metadata observed by the probe. The language used to express these constraints is the OCL language.

4.4.2 Custom Attributes

The designer can declare new attributes to store additional information about the state of the system at the time of the onset of the observed event. This is achieved by the probe derivation mechanism.

4.4.3 Predefined Operations

The two categories (elemental, composite) of the probe have the following predefined operations:

- **projection():** this is an operation that allows to definition the conditions of activation of a probe. Depending on the probe category, the projection() operation changes behavior:
  - In the case of an elementary probe, the projection() operation takes as a condition of activation the Boolean constraint specified by the filter attribute;
  - In the case of a composite probe, we associate the projection() operation of a state machine to express the assembly rules of the probes that form part of the composite probe.
- **activate():** this is an operation that runs automatically once the probe is activated, i.e., when:

- The filter selects the observed event for the elementary probes;
- The state machine associated with the projection() operation reaches the final state for the composite probes.

The main purpose of activate() operation is to call the update() method to update the probe attributes, and also unblock the transition blocked by the wait trigger.

- **update():** as mentioned above, the update() method is launched once the constraints of the projection are met. This is a method for updating probe attributes. If a probe is derived, the update() method must be redefined.

4.4.4 Custom Operations

The designer, by means of derivation, can customize the operations of the probes class used, to:

(1) Adapt to personal attributes added by derivation. In this case, the designer must bring up-to-date the update() operation so that it can take into account the update of the added attributes, as shown in the section dealing with probe derivation.

(2) Add new operations that will be performed when the probe is activated. In this case, the designer must add their declarations within the derived probe, then call them in the activate () operation.

5 Declaration and Instantiation of a Probe

The reporting of a probe is done independently of the system entities. That means a class of probes cannot be modeled as the usual classes and participate in associations with system entities.
Once the structure of the probe class is defined (a predefined probe or a new class created by derivation or composition), the designer can instantiate this class for use in their design models.

5.1 Case Study

To deal with this question, we have chosen to specify the behavior of a multivalent object by the notion of a state machine. To illustrate how to approach this question, we take the example of a car being repaired in a specialized agency. Because it is understood differently depending on the type of actor, the state of this car while it is being fixed can be thought of as multivalent (Figure 8). The breakdowns and repairs he must perform, the tools required to complete the repair, and the spare parts are all things a mechanic is interested in. A workshopManager, on the other hand, sees the repair from the logistical side, in the sense that he is interested in the assignments of repair lanes to carry out maintenance operations, in the scheduling of machinery, the distribution of spare parts, etc. For a client, the technical aspects of a repair are less significant than the specifics of the repair contract, the expenses involved, and the anticipated completion date. The agencyManager's interests are centered on the profitability of the repair, taking into account the real cost, the projected completion date, and the client contract that has to be formed.s

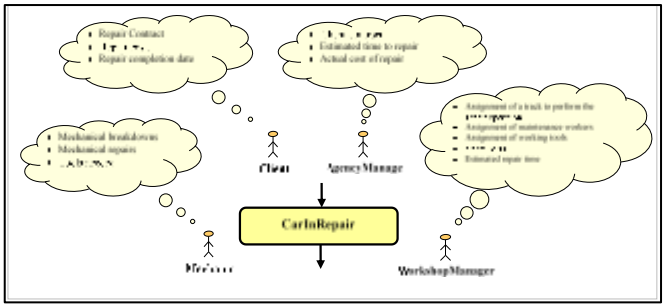

Fig. 8: Illustration of the CarInRepair Multi-View State

Figure 9 depicts an instance of the SignalSendProbe probe class in use. To track down signal transmissions, the SignalSendProbe-type ReparationOkObs probe is created in the model. The probe will activate whenever a signal is sent into the system if there are no limitations stated in the filter attribute; otherwise, the probe will choose events that satisfy the filter constraint (see the projection section below for more information). The pending items in a wait on the probe are unlocked and the probe characteristics are updated with the event parameters upon activation. Running the activate() function does this implicitly.

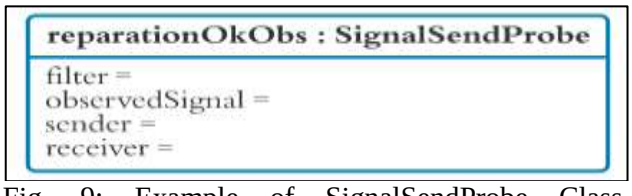

Fig. 9: Example of SignalSendProbe Class Instantiation

5.2 Probe Projection

Each form of probe is linked to a certain kind of event, thus if this kind of event occurs in the system, the probe becomes activated. Because we

occasionally need to filter events based on context information, this characteristic is not always desired. This amounts to applying additional constraints that must be satisfied when activating the probes. For example, if you take the two types of SignalSendProbe/SignalReceiveProbe probes, they allow you to reference the signal emissions/reception between running system objects. If no condition is specified in the probe definition, the probe will be triggered for any signal emissions/reception in the system. By applying the projection operation, it is possible to restrict the probe's field of activation to a particular context.

Assuming that emissions/reception are triggering events of the SignalSendProbe/SignalReceiveProbe probes; each time emissions/reception will appear in the system; the probes will be enabled after verification of the probe definition conditions

5.2.1 Principle of a Projection

We have equipped the probes with an operation-noted projection(). This operation allows additional constraints to be applied to the conditions under which a probe is activated. By defining requirements that must be met by the events targeted by the probe type, it specifies the context of the observation. The projection operation takes the constraints from the filter attribute of the probe. This attribute allows expressing in a string Boolean conditions on the data and metadata of observed events. To represent these restrictions, we employ OCL. An overview of the SignalSendProbe probe's projection is shown in Figure 10. Three OCL restrictions are combined to form the probe's filter, which will be verified by signal-sending events. The first restriction stipulates that the kind of observed signal must be of the ReparedCar type, the second that the transmitting object must be of the Mechanic type, and the third that the receiving object must be of the ResponsableAtelier type.

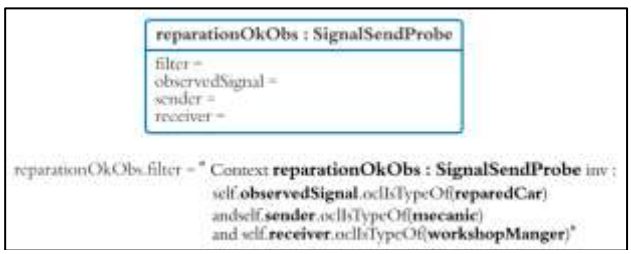

Fig. 10: Examples filter: projection of the SignalSendProbe probe

Only when the filter condition is fulfilled by the event parameters does the probe become active. The pending items in a wait on the probe are unlocked and the probe characteristics are updated with the event parameters upon activation. Running the activate() function does this implicitly.

5.2.2 Examples of Application of Projection

In this section, we present concrete examples of the application of projection on elementary probe types. Figure 11, Figure 12, and Figure 13 show the design model from which examples are drawn to illustrate the projection of probes. This is a class diagram taken from our case research "Management of a car repair agency". In order not to make the example of the probes complex, this class diagram is not developed according to the point-of-view approach. This is a simple diagram that represents part of the application structure but is sufficient to feed the examples presented in the rest of this session. This diagram is accompanied by two packages: the first contains the signal declarations and the second contains the probe declarations that will be used in the examples.

Examples of Filters Associated with SignalProbe probes

-Probe which detects any startReparation type signal emitted by the Agency Manager (Figure 14-a).
-Probe which controls any carRepared type signal emitted by a mechanic to a workshop Manager (Figure 14-b).
-Probe that controls any carRepared type signal emitted by a mechanic bearing the name of Patrick to the workshop Manager (Figure 14-c).
-Probe which detects all signal receptions (whatever their types) by Car type objects (Figure 14-d).

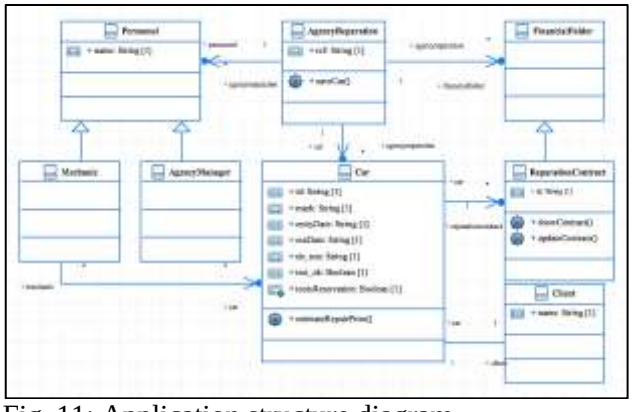

Fig. 11: Application structure diagram

Fig. 12: Signal declarations

Fig. 13: Probe declarations

(a) startRepObs.filter = "Context **startRepObs**: **SignalSendProbe** inv: self.**observedSignal**.oclIsTypeOf(**startReparation**) **and** self.**sender**.oclIsTypeOf(**Agency Manager**)"

(b) reparationOkObs.filter = "Context **reparationOkObs**: **SignalSendProbe** inv: self.**observedSignal**.oclIsTypeOf(**reparedCar**) and self.**sender**.oclIsTypeOf(**Mechanic**) and self.**receiver**.oclIsTypeOf(**WorkshopResponsable**)"

(c) reparationOkobs.filter = "Context **reparationOkObs**: **SignalSendProbe** inv: self.**observedSignal**.oclIsTypeOf(**reparedCar**) and if self.sender.oclIsTypeOf(**Mechanic**) **then let** m:Mechanic = self.**sender**.oclAsType(**Mechanic**) in m.**name** = "**Patrick**" **endif** and self.**receiver**.oclIsTypeOf(**WorkshopResponsable**)"

(d) carReceptSig.filter ="Context **carReceptSig**: **SignalReceiveProbe** inv: self.**recriver**.oclIsTypeOf(**Car**)"

Fig. 14: Examples of filters on SignalProbe probes

Examples of filters associated with OperationProbe probes

- Probe detecting all calls of the operation saveCar() of the class "RepairAgency" (Figure 15-a),
- Probe detects any return of the "evalutePrice" operation of the Car class (Figure 15-b).

(a) saveCObs.filter ="Context **saveCobs**: **OperationCallProbe** inv: self.**ownedObject**.oclIsTypeOf(**repairAgency**) and self.**observedOperation**.name='**saveCar**'"

(b) evaluePriceObs.filter ="Context **evaluePriceObs**: **OperationReturnProbe** inv: self.**ownedObject**.oclIsTypeOf(**Car**)

Fig. 15: Examples of filters on OperationProbe-type probes

Examples of filters associated with ObjectLifeProbe probes

- Probe to detect any instance creation of the Contract class (Figure 16-a).
- Probe detects the destruction of any Car-type object by the repairAgency (Figure 16-b)

(a) creationContractObs.filter = "Context **creationContractObs**: **ObjectCreationProbe** inv : self.**class**.name="**Contract**""

(b) destrCarobs.filter = "Context **destrCarObs**: **ObjectDestructionProbe** inv: self.**observedObject**.oclIsTypeOf(**Car**) self.**Killer**.oclIsTypeOf(**RepairAgency**)"

Fig. 16: Examples of filters on ObjectLifeProbe probes

Examples of filters associated with LinkProbe probes

- Probe for detecting car assignments to mechanics, by creating connections between Mechanic and Car objects (Figure 17-a),
- Probe that detects the destruction of the proprietary link between Customer and Car objects type (Figure 17-b).

(a) carAssignment.filter = "Context **carAssignment**: **LinkCreationProbe** inv: self.**observedAssociation**.name='**carAssignment**' and self.**associationEnd1**.oclIsTypeOf(**Machnic**) and self.**associationEnd2**.oclIsTypeOf(**Car**)"

(b) CarPropertyDestrobs.filter = "Context **CarPropertyDestrObs**: **LinkDestructionProbe** inv: self.**observedAssociation**.name='**carProperty**' and self.**associationEnd1**.oclIsTypeOf(**Client**) and self.**associationEnd2**.oclIsTypeOf(**Car**)"

Fig. 17: Examples of filters on LinkProbe probes

Examples of filters associated with AttributeChangeProbe probes

- Probe that detects the passing of testOK attribute to true of Car type objects (Figure 18).

testOkObs.filter = "Context **testOkObs**: **AttributeChangeProbe** inv: self.**object**.oclIsTypeOf(**Car**) and self.**attribute**.name='**testOk**' and (let m:Car = self.**object**.oclAsType(**Car**) in m.**testOk**=**true**) "

Fig. 18: Examples of filters on AttributeChangeProbe probes

Examples of filters associated with StateChangeProbe probes

- Probe that detects the entry into the OutOfOrder state of the Car objects (Figure 19-a).
- Probe detects any output from the Waiting state of Car objects (Figure 19-b).

(a) outOfOrderObs.filter = "Context **outOfOrderObs**: **StateEntryProbe** inv: self.**object**.oclIsTypeOf(**Car**) and self.**state**.name='**OutOfOrder**"

(b) exit WaitObs.filter = "Context **exitWaitObs**: **StateExitProbe** inv: self.**object**.oclIsTypeOf(**Car**) and self.**state**.name="**Waiting**"

Fig. 19: Examples of filters on StateProbe type probes

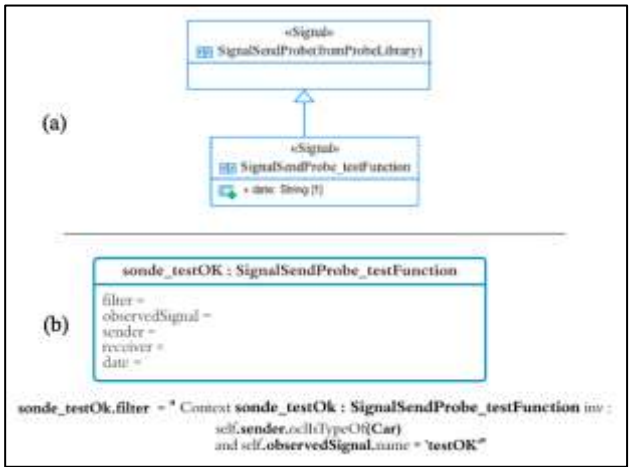

Fig. 20: SignalSendProbe Probe derivation Example

The metadata about the event that the probe witnessed is contained in the probes in the ProbeLibrary collection. When the probe is activated, the designer can specify new characteristics to hold more system status data. This is accomplished by determining the kind of probe being used.

Think about a SignalSendProbe probe type, which is going to be utilized to find the testOK signal that the Car objects are sending. To make this observation, use either a SignalSendProbe probe instance with the appropriate filter. The probe must be extended by an extra characteristic, such as date, if we wish to delay the delivery of the testOK signal at the moment the probe is activated. The SignalSendProbe_testFunction derivative probe's class (Figure 20-a) and an instance of it with its filter (Figure 20-b) are shown in Figure 20 below.

We remind you that the update() operation of a probe is the responsible operation for updating the attributes of the probe when it is activated. Therefore, to update the newly added properties (date in our case), this action has to be rewritten in the derived class.

The update() method of the SignalSendProbe_testFonction class is described in the Java code that follows.

```
Void update()
 {
        super.update();
        this.date = this.getSysDate();

        //getSysDate() méthode membre qui
        //retourne la date du système.
 }
```

6 Conclusion

In this paper, we have addressed the problem of behavioral specifications in the context of the VUML profile. We focused on the description of the event probe.

The VUML design approach is a globally decentralized point-of-view oriented modeling approach, which proceeds by partial developments of the application according to the subjective visions of the system actors. Modeling by part of the system offers advantages, especially in the case of complex systems, but this approach does not ensure independence in the development of the views. In fact, the coupling between the views can be important if the modeling of the current view requires information external to it. This situation makes the viewpoint-based design approach difficult to implement, if not impossible without altering the development of the other views in order to collect the missing information.

The definition of the view-object behaviors and the combination of these behaviors to create the overall behavior of the multi-view object are the two key issues that must be addressed in order to solve this challenge. The challenge of the issue is striking a balance between suggesting a strategy that provides the maximum freedom in the formation of perspectives while also supplying tools to support the fusion.

To address this issue, we provide brand-new methods that build on UML-based ones and are tailored to the unique characteristics of VUML.

This approach based on event probes solves the two problems mentioned above in the following way. Firstly, it allows to define the behavior of the

views independently of each other in the decentralized design phase. The designer does not have to worry, when declaring probes, about how the desired behavior will be expressed because this task is delegated to the declared probes. Secondly, the use of probes offers a simple principle for the composition of behaviors in the merging phase and avoids having to modify the view-models. Indeed, instead of modifying the view models to perform the composition, we proceed to a synchronization of the latter by acting on the declared probes. This consistency is achieved by finalizing the definition of the probes used in the different view models.

7 Future work

So far, we have talked about two techniques for personalizing basic probes. The first is the filter-based probe projection, which allows you to tailor a probe to a certain context by leveraging the data and metadata from the targeted events Second, when the probe is engaged, the probe derivation allows you to add new characteristics to record extra information about the system status. Nonetheless, these two processes allow only one sort of elemental probe to be customized at a time [28], allowing probes to act on just one type of event to be created, [22], [20], [23]. More complicated probes based on many types of events are not possible with either approach. So, the composition of elementary probes will be our next task, using some model transformation, [24], [26], [27], followed by the integration of the notion of probes in UML.

References:

[1] Ouali-Alami, Chaimae, Abdelali El Bdouri, Nisrine Elmarzouki, and Younes Lakhrissi. "View-based Modelling: Behaviour Specification based on UML Concept." (2022).

[2] Nassar, Mahmoud. "VUML: a Viewpoint oriented UML Extension." In 18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings., p. 373-376. IEEE, 2003.

[3] Nassar, Mahmoud, Adil Anwar, Sophie Ebersold, Bouchra Elasri, Bernard Coulette, and Abdelaziz Kriouile. "Code generation in VUML profile: A model driven approach." In 2009 IEEE/ACS International Conference on Computer Systems and Applications, pp. 412-419. IEEE, 2009.

[4] Rodríguez, Alejandro, Fernando Macías, Francisco Durán, Adrian Rutle, and Uwe Wolter. "Composition of multilevel domain-specific modelling languages." Journal of Logical and Algebraic Methods in Programming 130 (2023): 100831.

[5] Lakhrissi, Younes. "Integrating behavioral modeling into point-of-view design - Intégration de la modélisation comportementale dans la conception par points de vue." PhD diss., Universite Toulouse le Mirail-Toulouse II, 2010.

[6] Acher, Mathieu, Philippe Collet, Philippe Lahire, and Robert France. "Comparing approaches to implement feature model composition." In Modelling Foundations and Applications: 6th European Conference, ECMFA 2010, Paris, France, June 15-18, 2010. Proceedings 6, pp. 3-19. Springer Berlin Heidelberg, 2010.

[7] Chiriac, Noemi, Katja Hölttä-Otto, Dusan Lysy, and Eun Suk Suh. "Three approaches to complex system decomposition." In DSM 2011: proceedings of the 13th international DSM conference. 2011.

[8] Cavallaro, Luca, Elisabetta Di Nitto, Carlo A. Furia, and Matteo Pradella. "A tile-based approach for self-assembling service compositions." In 2010 15th IEEE International Conference on Engineering of Complex Computer Systems, pp. 43-52. IEEE, 2010.

[9] Guenov, Marin D., and Stephen Barker. "Requirements-driven design decomposition: A method for exploring complex system architecture." In International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, vol. 46962, pp. 145-151. 2004.

[10] Topper, J. Stephen, and Nathaniel C. Horner. "Model-based systems engineering in support of complex systems development." Johns Hopkins APL technical digest 32, no. 1 (2013): 419-432.

[11] Nassar, Mahmoud. "Viewpoint analysis/design: the VUML profile - Analyse/conception par points de vue: le profil VUML." PhD diss., 2005.

[12] Bennani, Saloua, Iliass Ait El Kouch, Mahmoud El Hamlaoui, Sophie Ebersold, Bernard Coulette, and Mahmoud Nassar. "A Formalization of Group Decision Making in Multi-Viewpoints Design." arXiv preprint arXiv:2004.14098 (2020).

[13] (text in French) Anwar, Adil. " IDM-based formalization of model composition in the VUML profile - Formalisation par une

approche IDM de la composition de modèles dans le profil VUML." PhD diss., Thèse de doctorat, Université de Toulouse, 2009.

[14] Anwar, Adil, Sophie Ebersold, Bernard Coulette, Mahmoud Nassar, and Abdelaziz Kriouile. "A Rule-Driven Approach for composing Viewpoint-oriented Models." J. Object Technol. 9, no. 2 (2010): 89-114.

[15] El Asri, Bouchra, Mahmoud Nassar, Bernard Coulette, and Abdelaziz Kriouile. "Multiviews components for information system development." In International Conference on Enterprise Information Systems, vol. 4, pp. 217-225. SCITEPRESS, 2005.

[16] Anwar, Adil, Sophie Ebersold, Bernard Coulette, Mahmoud Nassar, and Abdelaziz Kriouile. "Vers une approche à base de règles pour la composition de modèles. Application au profil VUML." Obj. Logiciel Base données Réseaux 13, no. 4 (2007): 73-103.

[17] El Asri, Bouchra, Mahmoud Nassar, Bernard Coulette, and Abdelaziz Kriouile. "Architecture d'assemblage dynamique de composants multivues dans VUML." In INFORSID, pp. 943-958. 2006.

[18] Bruneliere, Hugo, Florent Marchand de Kerchove, Gwendal Daniel, and Jordi Cabot. "Towards scalable model views on heterogeneous model resources." In Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, pp. 334-344. 2018.

[19] El Marzouki, Nisrine, Younes Lakhrissi, Oksana Nikiforova, and Mohammed El Mohajir. "The application of an automatic model composition prototype on the-Two hemisphere model driven approach." In 2017 International Conference on Wireless Technologies, Embedded and Intelligent Systems (WITS), pp. 1-6. IEEE, 2017.

[20] El Marzouki, Nisrine, Oksana Nikiforova, Younes Lakhrissi, and Mohammed El Mohajir. "Enhancing Conflict Resolution Mechanism for Automatic Model Composition." Appl. Comput. Syst. 19, no. 1 (2016): 44.

[21] Nikiforova, Oksana, Nisrine El Marzouki, Konstantins Gusarovs, Hans Vangheluwe, Tomas Bures, Rima Al Ali, Mauro Iacono, Priscill Orue-Esquivel, and Florin Leon. "The Two-Hemisphere Modelling Approach to the Composition of Cyber-Physical Systems." In ICSOFT, pp. 286-293. 2017.

[22] Chabibi, Bassim, Adil Anwar, and Mahmoud Nassar. "Towards a Model Integration from SysML to MATLAB/Simulink." J. Softw. 13, no. 12 (2018): 630-645.

[23] El Marzouki, Nisrine, Younes Lakhrissi, Oksana Nikiforova, and Mohamed El Mohajir, and Konstantins Gusarovs. "Behavioral and Structural Model Composition Techniques: State of Art and Research Directions." Transactions on Computers, WSEAS 16 (2017): 39-50.

[24] El Marzouki, Nisrine, Oksana Nikiforova, Younes Lakhrissi, and Mohamed El Mohajir. "Toward a generic metamodel for model composition using transformation." Procedia Computer Science 104 (2017): 564-571.

[25] Bennani, Saloua, Mahmoud El Hamlaoui, Mahmoud Nassar, Sophie Ebersold, and Bernard Coulette. "Collaborative model-based matching of heterogeneous models." In 2018 IEEE 22nd International Conference on Computer Supported Cooperative Work in Design ((CSCWD)), pp. 443-448. IEEE, 2018.

[26] Chabibi, Bassim, Abdelilah Douche, Adil Anwar, and Mahmoud Nassar. "Integrating SysML with simulation environments (Simulink) by model transformation approach." In 2016 IEEE 25th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), pp. 148-150. IEEE, 2016.

[27] Chabibi, Bassim, Adil Anwar, and Mahmoud Nassar. "Towards an alignment of SysML and simulation tools." In 2015 IEEE/ACS 12th International Conference of Computer Systems and Applications (AICCSA), pp. 1-6. IEEE, 2015.

[28] Anwar, Adil, Taoufiq Dkaki, Sophie Ebersold, Bernard Coulette, and Mahmoud Nassar. "A formal approach to model composition applied to VUML." In 2011 16th IEEE International Conference on Engineering of Complex Computer Systems, pp. 188-197. IEEE, 2011.

[29] Schützenmeier, Nicolai, Carl Corea, Patrick Delfmann, and Stefan Jablonski. "Efficient Computation of Behavioral Changes in Declarative Process Models." In International Conference on Business Process Modeling, Development and Support, pp. 136-151. Cham: Springer Nature Switzerland, 2023.

[30] Denysov, Viktor. "Software and information complex for district heat supply systems

modeling." System Research in Energy 1, no. 70 (2022): 38-45.

[31] Rehioui, Fadoua, and Abdellatif Hair. "Towards a Modeling approach based on Software Components." International Journal of Computer Applications 975 (2014): 8887.

Contribution of Individual Authors to the Creation of a Scientific Article (Ghostwriting Policy)

The authors equally contributed in the present research, at all stages from the formulation of the problem to the final findings and solution.

Sources of Funding for Research Presented in a Scientific Article or Scientific Article Itself

No funding was received for conducting this study.

Conflict of Interest

The authors have no conflict of interest to declare.