A New Approach for Multi-View Models' Composition using Probes Event

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

Abstract: - The paper presents the development and application of the VUML Probe profile, an extension of the VUML (View-based UML) approach to modeling complex software systems. It focuses on improving behavioral modeling by introducing probes for observing and monitoring events. In the introduction, the importance of separation of concerns in the management of large software systems is emphasized, with the introduction of view-based modeling and the VUML profile. The application context section presents the VUML analysis and design process, illustrated by a case study of managing an automotive repair shop. Probes are introduced as a modeling concept for event detection and control, with basic categories and methods for projection, derivation, and composition. The VUML Probe profile is presented, integrating probe stereotypes into the VUML meta-model, with conformance rules to maintain semantic consistency. The application of probes in the VUML process is demonstrated, in particular, to ensure the autonomous evolution of model-views. Abstract probes are defined during the composition phase, and then used in view models. Finally, related work and avenues for future research are discussed, including language enhancements, integration with aspect-oriented modeling, and tool development. In summary, the paper offers a comprehensive framework for integrating event observation mechanisms into the VUML approach, aimed at improving the modeling and management of complex software systems.

Key-Words: - View based modeling, VUML profile, VUML Probe_profile, event observation, multi-view states machine, behavior composition.

Received: July 14, 2023. Revised: February 14, 2024. Accepted: March 11, 2024. Published: May 20, 2024.

1 Introduction

The critical aspect in dealing with the intricacy of large software systems is the concept of "separation of concerns". This practice is indispensable to maintain manageability throughout the development process, the resulting models, and the code. Achieving separation of concerns can be executed through various methods, but the ultimate objective remains consistent: the identification of relatively autonomous "components" that can be allocated to different participants in the process. These components are then designed and constructed independently, with the ultimate goal of integrating them with minimal effort while ensuring future maintainability and adaptability.

The principal focus of this endeavor is on view-based modeling, [1], which is a variant of the object-oriented modeling approach tailored for the examination and design of expansive and intricate systems. This approach concentrates on the individuals who interact with the system and disassembles the specification by their specific needs. Consequently, the development of the VUML (View-based UML) UML profile was initiated, enabling the creation of a unique and shareable model that can be accessed based on each system actor's perspective. At the core of VUML lies the concept of a "multi-view class," comprising a foundational class that expresses the structural and behavioral characteristics shared among all system actors. Additionally, it consists of a set of views, each representing attributes pertinent to a particular actor. The connection between each view and its base is established through a newly stereotyped relationship termed "view extension." Dependencies among views are defined utilizing another stereotyped relationship named "view dependency," complemented by constraints articulated in plain language or OCL. An instance of a multi-view class takes the form of a "multi-view object."

However, it is important to note that the modeling of behavioral facets was not previously the main emphasis of the VUML profile's development efforts. In the absence of addressing

the behavioral aspects of views—how they react to invocations and how they collaborate to accomplish the behavior of multi-view objects—the VUML methodology primarily addresses the structural elements linked to view composition and data sharing.

In preceding research, the structural composition of models was addressed, [2]. Nonetheless, when delving into the composition of behavioral models, the issue becomes more intricate. A prior work, [3], [4] introduced the concept of "event probes" as a pivotal notion for addressing this challenge. This work revolved around the behavioral specification and composition of these object segments. It is widely acknowledged that this issue is more closely aligned with aspect composition than conventional interface-based composition. We illustrate that the novel behavioral specification architecture, grounded in event observation, yields favorable outcomes concerning slice connectivity and support for evolutionary design, particularly the addition of slices.

We introduced a modeling framework centered on event observation as the primary method for handling first-class object interactions. This was primarily achieved through the formalization of event concepts and their attributes. Additionally, we introduced a novel modeling concept known as a "probe," designed for event matching and the manipulation of event data.

The primary focus of this study is to provide further clarity on the aforementioned topics. We have developed new techniques, including the projection, derivation, and composition of probes, to enhance the precision of probe-related concepts. Our proposal is structured as a profile named the "VUML Probe profile."

The article is organized as follows: In Section 2, we establish the backdrop for our research, focusing on view-based modeling exemplified by the VUML profile. Additionally, we provide an illustrative example of its practical application. Section 3 delves into the fundamental concepts of our proposition, specifically examining probes and associated notions. Section 4 is devoted to introducing the VUML Probe Profile, while Section 5 investigates its implementation through a case study. In Section 6, we conclude the paper by juxtaposing our solution with current strategies, formulating conclusions, and outlining potential directions for future research.

2 Application Context: View-Based Modeling

According to various perspectives, the creation of the VUML profile, [5], aimed to address the requirements of modeling complex information systems using UML. Each perspective in the system reflects an actor's needs and privileges. These viewpoints can be viewed as practical, user-centric facets. A view is the result of generalizing a perspective across the entire system and applying it to an entity, typically a class.

The key idea in the VUML language revolves around the "multiview class." This class comprises a foundational class shared among all actors and a series of view classes, each tailored to a specific perspective and extending the foundational class. This construction of a multiview class facilitates accurate control of access permissions and the separation of concerns at the class level. To articulate the semantics of VUML, a meta-model, accompanied by constraint rules articulated in OCL, is employed.

2.1 The VUML Profile

VUML adopts a viewpoint modeling approach, also known as a viewpoint decomposition method, to organize and structure software system models. This approach recognizes that different actors involved in the development and use of a system have different perspectives on it. In this way, the system can be viewed from different angles, each highlighting specific aspects relevant to a particular interest group.

Here's how this method is applied in VUML:

- Identifying viewpoints: First, stakeholders identify the different viewpoints relevant to the system. These may include end-users, developers, testers, project managers, and so on. Each point of view highlights the aspects of the system that are important to the group concerned.
- System decomposition: Once the viewpoints have been identified, the system is decomposed into subsystems or logical components corresponding to each viewpoint. Each subsystem represents a part of the system that is relevant to a specific point of view.
- Viewpoint modeling: For each viewpoint, VUML models are created to represent the specific aspects of the system that are relevant. For example, a user viewpoint can be represented by use case diagrams describing the interactions between the system and its users,

while a design viewpoint can be represented by class diagrams describing the internal structure of the system.

- Viewpoint integration: Once the models for each viewpoint have been created, they are integrated to form a coherent overview of the system. This ensures that all aspects of the system are taken into account and that the needs and objectives of all stakeholders are met.

By using this point-of-view decomposition approach, VUML enables development teams to better understand system needs and requirements, taking into account the different perspectives of stakeholders. This facilitates communication, collaboration, and decision-making, all of which contribute to the success of the software project.

2.2 The VUML Analysis and Design Process

VUML aims to simplify the design complexity of software systems by leveraging fragmentation aligned with the system actors' needs and access rights. It achieves this by advocating for a method that facilitates model development at various levels of abstraction. This horizontal separation of concerns complements the vertical approach of MDA, [6].

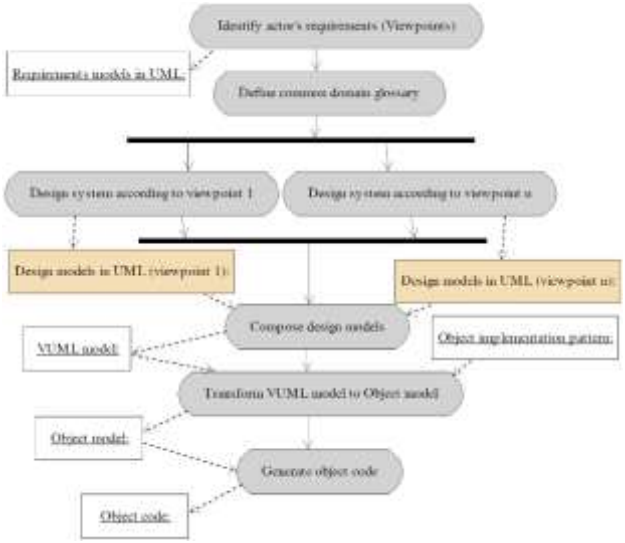


Fig. 1: Overall VUML process, [5]

The VUML design process consists of three main phases, each comprising several steps (refer to Figure 1). The initial step involves identifying the actors' needs, with the primary goal being the development of a requirements model, typically represented as a UML use case diagram. In the decentralized second phase of the process, various PIM models, [7] are generated, each representing a distinct viewpoint. In this stage, a set of UML models is produced, comprising class diagrams, state machines, sequence diagrams, and additional elements. The final step entails a composition operation, where autonomously developed design models are merged to form a comprehensive VUML design model.

2.3 Case Analysis

To illustrate our approach, we will use a case study involving the management of an auto repair shop, as depicted in Figure 2. This case study involves a complex information system with multiple actors, including a manager, technicians, clients, and more. It serves as an example to showcase how the VUML approach described earlier is applied in practice. The study primarily focuses on specifying object behavior in the second phase and then combining these behaviors in the fusion phase, which is the third phase of the process.

For simplicity, we will limit our examination to the following actors and their associated activities:

- The customer or automobile owner (Client): The system's functionality should allow customers to access information about their cars, track their vehicles' repair progress, and review repair and expert reports.
- The mechanic (Mechanic): Mechanics should be able to review the history of failures, make amendments, and record expert and repair reports, as well as log information about the parts that have been repaired.

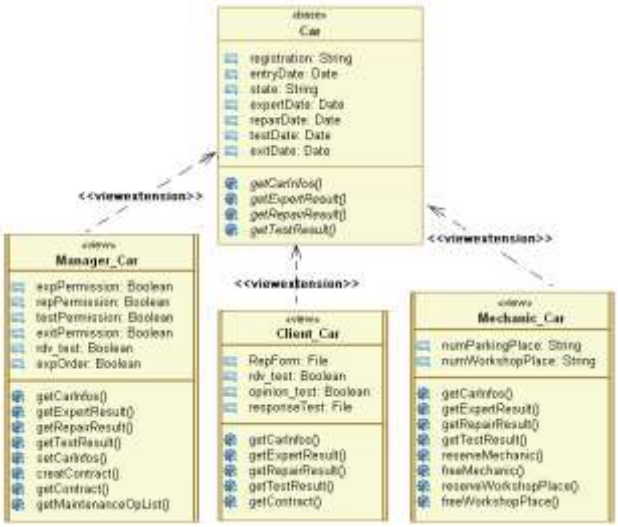


Fig. 2: Segment from the VUML class diagram: the multi-view class "Car"

The result of merging various structural models that were previously distinct is a VUML model that incorporates multiview classes. These classes are formed by combining homonymous classes created independently. An excerpt of a VUML class

diagram depicting the multiview class "Car" is provided in Figure 2. This class includes a "base" section, stereotyped as such, accessible to all actors. Additionally, it contains "views," stereotyped as "view," representing parts specific to each actor (Client, Manager, Mechanic).

3 Probes

A probe is a modeling construct used to detect and control events that are important for achieving a specific objective. We utilize probes in particular to simulate implicit interaction between objects. The existing modeling elements, particularly in UML, cannot be used to concretize the goals of the probe event notion. We lay out the foundational ideas for comprehending our proposal in this part.

We have determined groupings of probes that correspond to the typical execution-related event types. We may hone these basic sorts of probes to describe a specific behavior using the notion of projection. We outline the probe derivation process, which enables the creation of new probe classes with supplementary properties while consistently observing the same event type as the parent class. We offer a second, more difficult notion that makes use of the composition mechanism to customize a class of probes. Composite classes can simultaneously observe various events.

3.1 Basic probe categories: Probe Library

After a thorough examination of various events occurring during system usage (as illustrated in Figure 3), we have identified three primary categories of fundamental probes. By MDA terminology, these basic probe types are situated at the M1 modeling level, [8].

These basic probes are defined within a ProbeLibrary, providing system designers with the option to employ them as preconfigured classes in their models. By instantiating each library type, concrete probes can be created. As mentioned earlier, these probe types can be tailored to specific contexts through principles such as projection, derivation, and composition.

Distinguishing these probes is their capability to manage data either at the model level (M1 level) or the meta-model level (M2 level). For example, the class type designates its membership class, an element at the M2 level, while the ObjectProbe type oversees the observed Object type, an object at the M1 level. Achieving this requires employing a language that facilitates reflexivity.

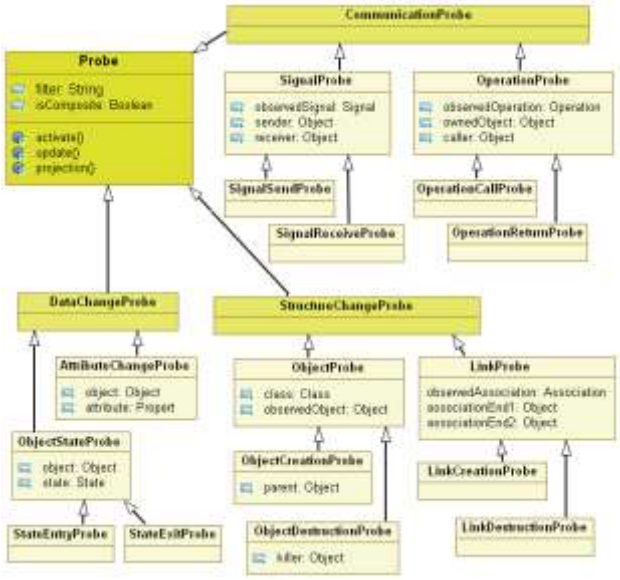


Fig. 3: Basic probe categories: ProbeLibrary

3.2 Declaration and Instantiation of Probes

Probes are declared independently of system entities, following a similar process to traditional class declarations. Once the structure of a probe class is defined, developers can instantiate these probe classes and incorporate them into their design models. As depicted in Figure 4(a), we present an example of the SignalSendProbe class instantiation. To monitor signal transmissions, the model instantiates the probe repairOkObs. The probe becomes active for any signal transmission within the system when a constraint is specified in the filter attribute. Otherwise, the probe will choose events conforming to the constraint filter, as detailed in the subsequent section on projection.

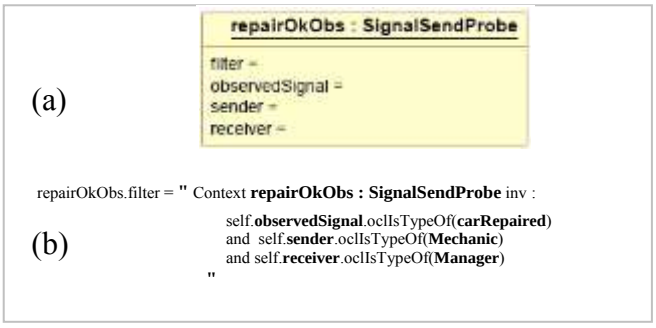


Fig. 4: (a) An instance illustration of a probe (b) Exemplary filter

3.3 Utilizing Probes in UML

In the following example, we illustrate the application of the probe approach in UML development. Let's assume we have a class named "MyClass," which intends to utilize the probe concept to monitor when the agency manager sends

the "startRepair" signals. To implement this observation, we proceed with the following steps:

1. Instantiate the probe class associated with the observed event type, specifically the "SignalSendProbe" type. The resulting instance is labeled as "startRepObs." This is accomplished by defining an "InstanceSpecification" UML element and assigning the stereotype "probe" to it.
2. Define the constraints that need to be evaluated at the observation moment to specify the probe filter. In our case, we have two constraints applied to the "observedSignal" and "sender" properties.
3. Declare a reference to the instantiated probe by introducing a UML "Reception" within the class and subsequently annotating it as «probeUse».
4. Incorporate the probe instance within the behavioral description of the class, particularly within the state machine compatible with the class.

A summary of the "startRepObs" probe used to describe the behaviors of "MyClass" can be found in Figure 5. The definition of the "startRepObs" probe is presented in Figure 5-top, while Figure 5-low illustrates how the state machine associated with the "MyClass" class references and utilizes this probe.

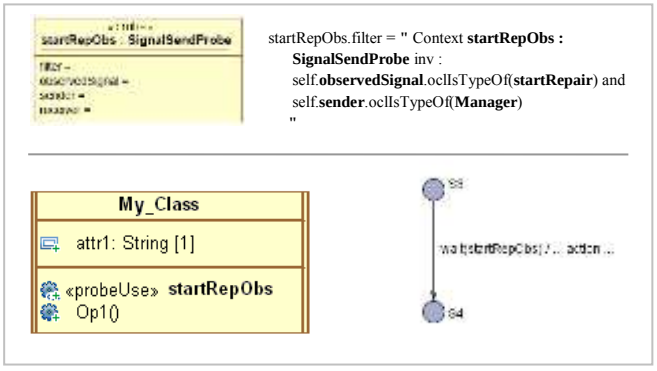


Fig. 5: Utilizing probes in UML development

3.4 Probes Projection

Each type of probe is assigned to a specific category of event, causing the probe to be triggered whenever an event of that particular type occurs within the system. However, the use of the "filter" attribute is not obligatory since there are instances where events need to be filtered based on contextual information. To apply additional constraints for such cases, these limitations must be defined and met when the probe is activated.

The projection process allows for the imposition of additional criteria on probe activation. It involves adding prerequisites that events must adhere to in order to match the probe type, thus establishing the context for observation. This is achieved through the "filter" attribute, which enables the expression of Boolean data and metadata conditions of observed events in a character string. These constraints are expressed using the OCL programming language.

Figure 4(b) illustrates the projection of the SignalSendProbe probe. The filter depicted in this figure amalgamates three OCL constraints that will be validated against events of the signal transmission type. The initial constraint mandates that the observed signal pertains to the "carRepaired" type, the second necessitates the transmitting object to be of type "Mecanicien," and the third specifies that the receiving object must be of type "Manager."

The probe is activated solely when an event's parameters satisfy the filter conditions. Upon activation, all attributes of the probe are automatically updated to reflect the event's parameters, and the transitions that follow the probe's activation become executable within the state machines used by the objects employing the probe. This process is triggered automatically when the operation is activated.

3.5 Probe Derivation

The previously defined probe classes within the ProbeLibrary encompass the metadata of the observed event type. Developers have the flexibility to declare an additional attribute to store supplementary data concerning the system's status and the activation time of the probe. This customization is contingent upon identifying the specific probe under consideration.

Consider a SignalSendProbe, which is designed to identify the "testOK" signal sent by objects of the "Car" type. Now, let's introduce a specific instance of the SignalSendProbe probe, equipped with a filter tailored for this observation. If we require the precise moment when the signal was sent at the time of probe activation, we can enhance this probe with an additional attribute, such as "date." The class for the derived probe, known as "SignalSendProbe_testFunction," is depicted in Figure 6(a), along with an instance of this class and its corresponding filter in Figure 6(b).

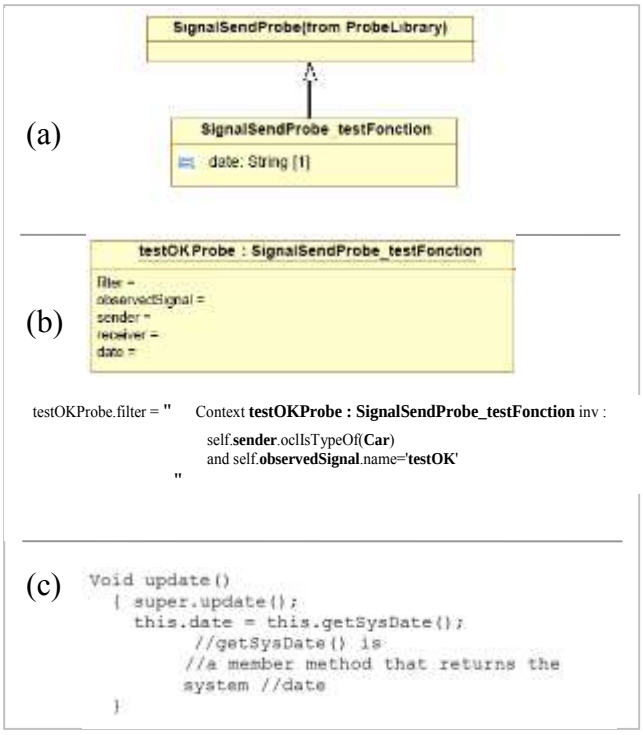


Fig. 6: Probe derivation: example of testOkProbe

It's worth noting that the "update()" action of the probe is responsible for refreshing the probe's properties upon activation. Consequently, to ensure that the newly introduced attributes, like "date" in our example, are properly updated, this operation needs to be redefined in the derived class. The Java code for the "update()" operation of the "SignalSendProbe_testFunction" class is presented in Figure 6(c).

3.6 Combining Basic Probes

We have discussed two methods for modifying basic probes thus far. The first method is projection, which employs a filter to customize a probe according to a specific situation using data and metadata events (see Section III.D). The second method is probe derivation, which offers the option to introduce new attributes for storing additional data related to the system's state at the moment the probe is activated (see Section III.E). However, it's essential to note that these two processes allow customization for only one type of probe at a time. These approaches are not suitable for handling complex probes involving multiple event types. In this section, we introduce a third mechanism called probe composition, aiming to simultaneously monitor various types of occurrences.

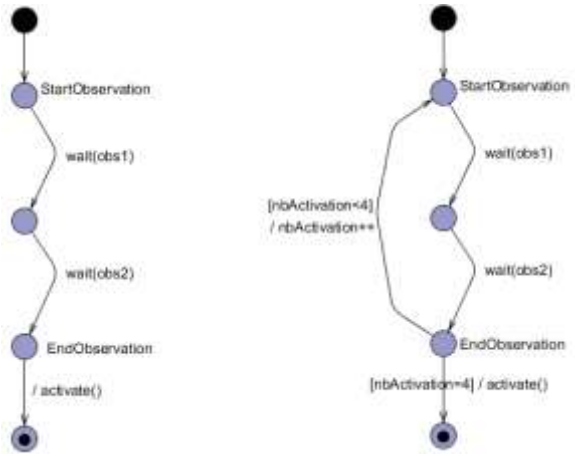
To represent the arrangement of probes, we have chosen to utilize a state machine. This decision was made for several reasons, including:

Simplifying the designers' work by expressing composition visually rather than through mathematical notation.

Representing the timing and sequencing of system events.

Leveraging the capabilities and resources provided by a state machine in describing behaviors in a state/transition format.

Examine a straightforward scenario where we need to reference the occurrences of "A behavior followed by B behavior." Suppose two probes, obs1 and obs2, record these behaviors, with obs1 capturing A and obs2 capturing B. The composition of these two probes, obs1 and obs2, represented as a state machine, is depicted in Figure 7(a).



(a) (b)

Fig. 7: Composition probe examples

The state machine illustrating the composition is presented in Figure 7(b). To keep track of the number of sequence realizations, a variable named nbActivation, initially set to 1, is employed in the same example. In this scenario, the probe is intended to be triggered after four sequences, which consist of the "A behavior" followed by the "B behavior." It's important to note that any scenario associated with defining the constructed probe can be inserted between the two states: "startObservation" and "EndObservation." Once the "EndObservation" state is exited, the "activate()" operation is executed, marking the conclusion of the observed behavior.

4 Probe for VUML Profile

The VUML Probe Profile incorporates various stereotypes, namely ProbeClass, Probe, ProbeUse, ProbeEvent, and Wait, into the VUML metamodel. These stereotypical designations result from the mapping process applied to the comprehensive

meta-model detailed in [9]. A summary of the abstract syntax associated with the proposed profile can be observed in Figure 8.

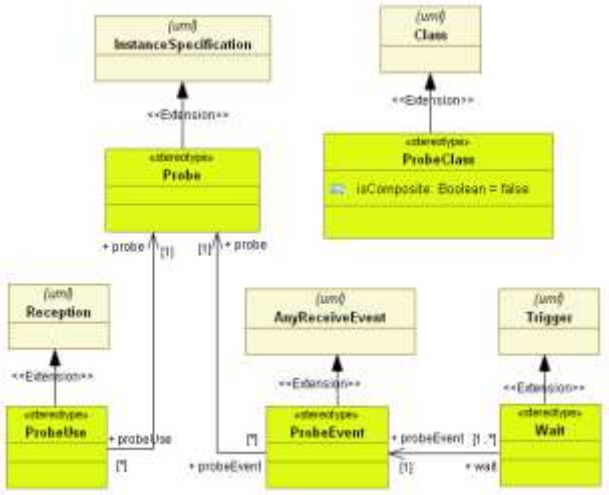


Fig. 8: Probe_Profile definition OF stereotypes

- The meta-model for components aims to create a linkage between the domain concepts encapsulated by our profile and the corresponding UML concepts. In this segment, we introduce the translations we have selected for the stereotypes associated with the proposed meta-model elements:
- The UML Class metaclass is expanded with the stereotype "probeClass." It serves as a representation of the components of the probe library type. The tagged value "isComposite" helps differentiate between the two probe types (elementary and composite). The default value "false" is assigned to newly created items, indicating the elementary probe class.
- The UML InstanceSpecification metaclass is enriched with the stereotype "probe." It identifies objects that are instances of classes stereotyped with « probeClass ». We have established OCL constraints to ensure this, as detailed in the subsequent section.
- The UML Reception metaclass is broadened with the stereotype "probeUse." In UML, a reception is a modeling element specifying the class intending to use a signal. This metaclass is extended because it closely aligns with the semantics of ProbeUse. In terms of syntax, a reference to a probe is indicated as « probeUse » obs1.
- The UML AnyReceiveEvent metaclass is improved with the stereotype "probeEvent."
- The UML Trigger metaclass is complemented with the stereotype "wait." The syntax "wait(obs)" in a state-machine transition signifies that the Wait trigger is awaiting the activation of the probe "obs" to traverse this transition.

4.1 Compliances Rules

In this section, we offer semantic guidelines concerning the components within the Probe profile. Initially, these guidelines are presented in natural language before being translated into OCL.

- Every InstanceSpecification labeled with the stereotype "probe" must also possess a classifier marked with the stereotype "probeClass."

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

- A trigger stereotyped as « wait » cannot used to describe an event.

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

- Only elements of the type Probe can be referred to by all elements with the type Reception and the stereotype "probeUse".

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

5 Application

Ensuring the autonomy of model-view evolution is a fundamental principle in the multiview design approach. However, this principle may be compromised when there is a substantial

interdependence between the behaviors of the model-views. To address this challenge, we have proposed an observation approach based on event probes.

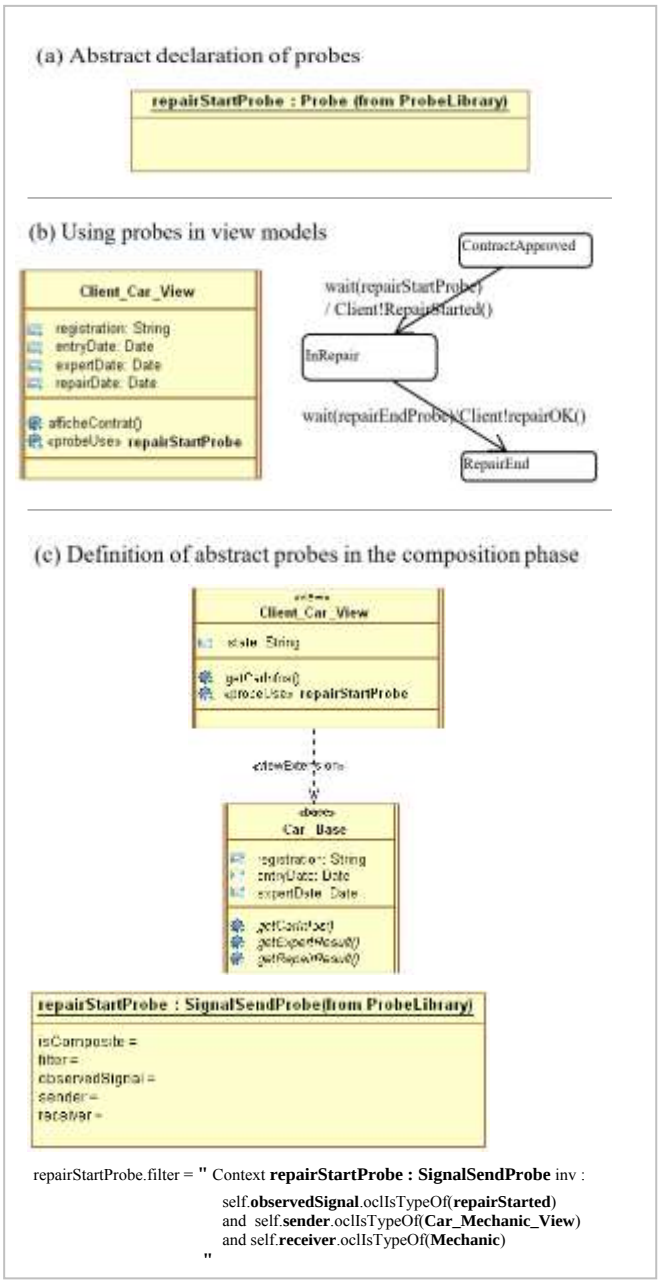


Fig. 9: Using probes in VUML process

For instance, consider the scenario where we need precise information about when the car repair process begins while developing the state machine for the client actor. However, the client model view lacks direct access to this information, and it remains unclear which actor is responsible for initiating the repair process (whether it's the agency manager, the technician, or the workshop manager). Furthermore, the specific representation of this activity is yet to be determined (whether it involves a signal exchange, a message, or an attribute change).

The solution involves employing probes to describe the behaviors of the views independently of each other, and the principles of behavior composition during the composition phase are detailed in the following sections (Section 5.1).

5.1 Abstract Definition of Probes

When examining the state machine that represents the car within the client view, it becomes imperative to determine the point at which the car repair operation commences.

However, the triggering event for this initiation remains unidentified, contingent upon another viewpoint that might not have been explicitly modeled. This event will only be established during the composition phase of viewpoints. Consequently, we introduce an abstract probe termed "repairStartProbe." Figure 9(a) illustrates the abstract declaration of this probe, presented as a direct instance of the Probe class, which serves as the root class in the ProbeLibrary.

5.2 Employing Probes in View Models

The designated probe is employable in the behavioral delineation of the client perspective, particularly within the behavioral specification of the Car class. To incorporate it, a probe reception must be established within the Car Client View class, indicated by the « probeUse » stereotype. Subsequently, the probe can be employed in the behavioral specification for this class through its state machine.

In Figure 9(b), one can observe the application of the "repairStartProbe" probe, instigating the transition between the "ContractApproved" and "InRepair" states.

5.3 Defining Abstract Probes during the Composition Phase

As mentioned in [9], the behavior composition seeks to specify the behavior of multi-view reactive objects after the structural composition is finished. These multi-view objects are made up of multiple parts, each of which expresses a view behavior that was defined during the view modeling stage using a state machine. Either the definition of abstract parameters or the concretization of abstract probes allows for the mapping and synchronization among view machines.

Figure 9(c) depicts the result of the model composition process conducted during the decentralized phase. An exemplar from the Car

class, after structural amalgamation with a solitary actor (the client), is delineated in Figure 9(c)-top. The Car is exhaustively delineated by the foundational class "CarBase," shared uniformly across all perspectives, along with an assemblage of perspectives, each explicating the attributes relevant to a particular actor.

In Figure 9(c)-low, the instantiation of the "repairStartProbe" probe is illustrated after the antecedent parameter concretization that was left unspecified. The "RepairStartProbe" pertains to the SignalSendProbe category and monitors the "repairOrder" signal.

6 Related work

This research builds upon the findings presented in a previous work [6], which introduced the concept of a probe and its application in specifying behavior and facilitating communication with other object slices, accompanied by illustrative examples. The current study extends these insights by offering enhanced methodologies for composing and refining event probes, along with an exploration of their integration into the VUML profile.

Prior approaches to event observation have been employed in diverse domains, including formal verification, software testing, debugging, and various aspect-driven modeling techniques. To our knowledge, object-oriented modeling languages have not been previously explored for event observation, and there have been no proposals for modeling languages that incorporate probes, event types, or related concepts.

In the realm of formal verification, the concept of event observation was first introduced as a method for formally specifying attributes in the Véda verification tool, [10]. This idea was subsequently extended to tools for verifying asynchronous concurrent systems, as evidenced in [11], [12]. While our work draws inspiration from these earlier efforts, it distinguishes itself by integrating the concept within a robust modeling language like UML. This enables the modeling of event types, including meta-parameters, as well as refinement and composition processes. Notably, our approach innovatively positions event observation as a primary method for modeling object interactions.

In synchronous models, where specific requirements for a language or method to model observation are absent due to the limited occurrence of event types (primarily data changes), our approach introduces a novel perspective within the UML framework. The exploration of decentralized supervisory control in discrete-event systems, as seen in works like, [13], has extensively examined the computational expressiveness of (distributed) event observation but without associating it with a particular language for expression.

Within the context of aspect-oriented programming, methodologies suggesting the specification of joint points through event sequences, [14], [15], [16], [17] exist. Our approach aligns with the Concurrent Event-Built AOP (CEAOP) computational model, described in [18], sharing similarities in concepts such as parallel system components, aspect composition, and event-based synchronization. Nevertheless, a fundamental distinction lies in the fact that our approach delves into the properties of events and introduces the probe construct, whereas, in CEAOP, events are treated as plain (uninterpreted) synchronization labels.

In the domain of distributed event-based systems (DEBS), specific efforts have been made to define a language for event detection, as evident in [19]. The key difference between our approach and [19] lies like their language, which is textual and not integrated into a general-purpose modeling language like UML. Moreover, their focus on large-scale event stream processing differs significantly from our scope, necessitating the use of distinct languages. Other instances of event observation can be found in fields such as autonomous agents, [20] and program tracing and debugging, exemplified by technologies like Sun's D-Trace.

7 Conclusion and Future Research Paths

This paper introduces a UML profile that elevates event observation to the status of a primary object interaction mechanism while also extending the existing VUML profile dedicated to view-based modeling. Our profile provides mechanisms for composing and refining event probes.

In this proposal, we have opted for an approach based on adding new, specific mechanisms to VUML for modeling and composing view-object behaviors. To this end, we have introduced the notion of event probes to specify implicit communications between view objects through event observations. This makes it possible to decouple specifications that are a priori strongly interconnected, design them separately, and then integrate them without having to modify them. To achieve our goal, we first defined the concept of an event probe, identified the different types of probes

with their associated parameters, [4], and then defined a set of concepts for enriching and manipulating probes.

System part, [21], modeling with VUML offers advantages for complex systems, but has two main limitations:

- Views need to be developed with an eye to the dependencies between them. Coupling between views is crucial, as each view requires external information, which can make the view-based design difficult to implement. Development coordination is needed to share the information required to integrate views.
- Model merging is complex and often requires human intervention, despite attempts at automation.

In conclusion, although the two approaches to model composition - UML-based composition, [22] and the event probe approach - aim to provide an efficient and accurate representation of complex systems, they differ in their approach and focus.

The UML-based approach focuses on static system modeling using diagrams and relationships defined in UML. This enables a detailed representation of the system's structure and behavior through a variety of views. However, it can suffer from the complexity of managing dependencies between views, which can make development coordination difficult.

On the other hand, the event probe approach focuses on dynamically monitoring interactions between system components by capturing and analyzing events. This provides an in-depth understanding of the system's real-time behavior but can be more difficult to model exhaustively and integrate into a global representation.

Future work. Numerous avenues for future research remain open, encompassing tools, methodologies, and language enhancements. On the language front, one potential avenue involves simplifying the distinct specification of viewpoints based on abstract probes. These abstract probes could potentially be treated as template parameters, and more advanced UML methods, [23], such as templates, might be leveraged for this purpose. By instantiating templates with concrete probes, we could facilitate view integration.

View-based modeling and aspect-oriented modeling (AOM) share certain similarities, as previously mentioned. However, the solution outlined in this paper may not be entirely suitable for AOM, primarily because aspect specifications sometimes require intrusive actions, such as limiting or modifying the behavior of the base model to which an aspect is applied. The mechanisms presented in this paper may not fully accommodate such requirements. Therefore, our future work will revolve around expanding the current concept into a comprehensive AOM language.

Another promising direction for future research involves the incorporation of high-level inter-object behavior specifications into VUML. Currently, the profile predominantly focuses on state machines and techniques for specifying intra-object behavior. Expanding the scope to encompass inter-object behavior specifications represents a significant area of interest.

Lastly, it's important to note that the prototyping tool described in [11], should be regarded as a proof-of-concept on the front of the tool. To establish the scalability of our approach for realistic models, it becomes necessary to extend this tool into a fully functional code generator. Furthermore, integrating this tool into a unified platform, along with various other tools developed around VUML (such as [5]), is a critical step forward in our research agenda.

References:

[1] Bruneliere, H., Burger, E., Cabot, J., & Wimmer, M. (2019). A feature-based survey of model view approaches. *Software & Systems Modeling*, 18(3), 1931-1952.

[2] El Marzouki, N. (2021). *Composition of models in multi-modeling approaches based on Model-Driven Engineering* (Composition des modèles dans les approches de multi-modélisation basée sur l'Ingénierie Dirigée par les Modèles (Doctoral dissertation)), Universite sidi mohammed ben abdellah.

[3] Ouali-Alami, C., El Bdouri, A., and Lakhrissi, Y. (2023). Proposition of the Probe-Event Approach for View-Based Modeling, 20, 206-219, https://doi.org/10.37394/23209.2023.20.24

[4] Li, C. S., Darema, F., & Chang, V. (2018). Distributed behavior model orchestration in cognitive internet of things solution. *Enterprise Information Systems*, 12(4), 414-434.

[5] Anwar, A., Ebersold, S., Coulette, B., Nassar, M., & Kriouile, A. (2010). A Rule-Driven Approach for composing Viewpoint-oriented Models. *J. Object Technol.*, 9(2), 89-114.

[6] El Hamlaoui, M., Ebersold, S., Bennani, S., Anwar, A., Dkaki, T., Nassar, M., & Coulette, B. (2021). A Model-Driven Approach to align

Heterogeneous Models of a Complex System. *The Journal of Object Technology*, 20(2), 1-24.

[7] Essebaa, I., & Chantit, S. (2017). QVT transformation rules to get PIM model from CIM model. *In Europe and MENA Cooperation Advances in Information and Communication Technologies* (pp. 195-207). Springer, Cham. https://doi.org/10.1007/978-3-319-46568-5_20

[8] Kim, W. Y., Son, H. S., Park, Y. B., Park, B. H., Carlson, C. R., & Kim, R. Y. C. (2008). Automatic MDA (model driven architecture) transformations for heterogeneous embedded systems. *Proceedings of the 2008 International Conference on Software Engineering Research and Practice, SERP 2008*, pp.409 – 414. Las Vegas, Nevada, USA

[9] Lakhrissi, Y. (2010). *Integrating behavioral modeling into point-of-view design* (Intégration de la modélisation comportementale dans la conception par points de vue (Doctoral dissertation)), Universite Toulouse le Mirail-Toulouse II.

[10] Renuka, G., 2023. Monitoring the state of materials in verification environment for IP architectures using python-based verification mechanism. *Materials Today: Proceedings*, 81, Part 2, pp.761-770, https://doi.org/10.1016/j.matpr.2021.04.233.

[11] Lalanne, F., Maag, S., De Oca, E. M., Cavalli, A., Mallouli, W., & Gonguet, A. (2009, November). An automated passive testing approach for the IMS PoC service. *In 2009 IEEE/ACM International Conference on Automated Software Engineering* (pp. 535-539). IEEE. Auckland, New Zealand, https://doi.org/ 10.1109/ASE.2009.33.

[12] Ahmad, M., Belloir, N., & Bruel, J. M. (2015). Modeling and verification of functional and non-functional requirements of ambient self-adaptive systems. *Journal of Systems and Software*, 107, 50-70, https://doi.org/10.1016/j.jss.2015.05.028.

[13] Pham, M. T., & Seow, K. T. (2011). Discrete-event coordination design for distributed agents. *IEEE Transactions on Automation Science and Engineering*, 9(1), 70-82.

[14] Hannousse, A. (2019). Dealing with crosscutting and dynamic features in component software using aspect-orientation: requirements and experiences. *IET Software*, 13(5), 434-446.

[15] Van Ham, J. M. (2015). *Seamless concurrent programming of objects, aspects and events* (Doctoral dissertation), Ecole des Mines de Nantes.

[16] Asteasuain, F., & Braberman, V. (2017). Declaratively building behavior by means of scenario clauses. *Requirements Engineering*, 22(2), 239-274.

[17] Wong, P. Y., Bubel, R., de Boer, F. S., Gómez-Zamalloa, M., De Gouw, S., Hähnle, R., & Sindhu, M. A. (2015). Testing abstract behavioral specifications. *International Journal on Software Tools for Technology Transfer*, 17(1), 107-119.

[18] Jaylet, T., Coustillet, T., Jornod, F., Margaritte-Jeannin, P. and Audouze, K., 2023. AOP-helpFinder 2.0: Integration of an event-event searches module. *Environment International*, 177, p.108017, https://doi.org/10.1016/j.envint.2023.108017.

[19] Chen, X., & Li, Q. (2020). Event modeling and mining: a long journey toward explainable events. *The VLDB Journal*, 29(1), 459-482.

[20] Horman, Yoav, and Gal A. Kaminka. "Improving Sequence Learning for Modeling Other Agents." *Proceedings of the AAMAS 2004 Workshop on Learning and Evolution in Agent-Based Systems*. 2004. Tulsa, Oklahoma, USA.

[21] Kühne, T. (2022). Multi-dimensional multi-level modeling. *Software and Systems Modeling*, 21(2), 543-559.

[22] Shirole, M., & Kumar, R. (2013). UML behavioral model-based test case generation: a survey. *ACM SIGSOFT Software Engineering Notes*, 38(4), 1-13.

[23] Besnard, V., Teodorov, C., Jouault, F., Brun, M., & Dhaussy, P. (2021). Unified verification and monitoring of executable UML specifications. *Software and Systems Modeling*, 20(6), 1825-1855.

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 conflicts of interest to declare.