Scenario-aware dataflow

Theelen, B.D.; Geilen, M.C.W.; Stuijk, S.; Gheorghita, S.V.; Basten, A.A.; Voeten, J.P.M.; Ghamarian, A.H.

Published: 01/01/2008

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the author's version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 05. Dec. 2018
UML Profile for Modeling System Observation

Mathias Funk, Piet van der Putten and Henk Corporaal
Abstract

Nowadays interactive electronics products offer a huge functionality to prospective customers, but often it is too huge and complex to be grasped and used successfully. In this case, customers often abort the struggle and return the products to the shop. Also the variability in scope and features of a product is so large that an up-front specification becomes hard if not impossible. To avoid the problem of an inadequate match between customer expectations and designer assumptions, new sources of product usage information have to be developed. One possibility is to integrate observation functionality into the products, continuously involving real users in the product development process. The integration of such functionality is an often overlooked challenge that should be tackled with an appropriate engineering methodology. This report presents on-going work about a novel design for observation approach that supports early observation integration and enables the cooperation with various information stakeholders. We show how observation can be embedded seamlessly in a model-driven development process using UML.

1 Introduction

Complex innovative electronic products often fail to satisfy customers’ needs. Products are too complicated and the inherent functionality is often not relevant to user needs and expectations. Increasing numbers of returned products support this [1]. On the other hand, nowadays products are hard to specify because of their high complexity and because of rapidly changing user demands. Also, faster cycles in the product creation process cannot benefit from traditional feedback channels any more. While, a couple of years ago, a product technology could reach maturity within 10 to 20 cycles, thus allowing for gradual improvements, today’s products have to accomplish the same within three cycles. Obviously, delivering a mature product in this setting becomes difficult.

Accordingly, complex interactive products should be built for rapid changes. Products can be adapted to changing needs during development and even after release, in terms of firmware updates and the like. Still, targeting the product for a certain user base is a major problem in the industry [1]. One reason for this is the lack of usage information which is reliable enough to base the further development of the product on.

Our approach towards this problem is to build observation modules into products. These observation modules can be configured remotely and observe parts of the system including user interaction, system performance and potentially user satisfaction with the system provided functionality. The products are given to selected key testers who use the products in their habitual environment - something, which promises to yield more representative data than, for instance, usability labs. It is important to note that even during the experiment, that is, with the products residing at the tester’s home, the configuration of observation can be changed according to recent findings of information stakeholders. The observation modules send collected data to a server on the Internet. There, the data is aggregated and made accessible for analysis purposes which might be a real-time visualization of usage statistics or a more extensive analysis by means of process mining tools that capture patterns inside the data stream.

This report concentrates on the integration of such observation facilities into products. We propose a model-driven technique to do this in an efficient and structured way which is tailored to current system
development practices. Subsequently, we show how observation-related parts of the system can be modeled by means of a novel UML profile. The report ends with a conclusion and an outline of possible future steps.

2 Related work

The remote monitoring of products has been done before, ranging in scope from the monitoring of cars to building automation, computer programs, mobile devices, and websites [6, 7, 8, 11]. However, our research is different in two important aspects: First, in our approach we assume that information stakeholders are not willing to use complex programming paradigms to achieve the sought-after data, therefore we use a visual language to specify observation behavior in a domain-specific way. Second, the integration of observation functionality into the target system is described in a software engineering process which is, in our opinion, necessary for widespread use. On the technical level we rely on the proven model-driven engineering approach, but also try to apply more agile modeling techniques like model interpretation [3] that allows for dynamic adaptation of runtime systems without the need for client compilation support. The modeling of observation systems is performed using the Unified Modeling Language (UML) [10] and, more precisely, its profile extension mechanism [2].

Besides that, there is also the area of large enterprise event reporting systems like WBEM [12]. These systems aggregate business information from various sources and present this data to stakeholders. The general approach of these systems is similar to that of an observation system, however the proposed observation system aims not only at scalability, but is also flexible and light-weight. Since observed products might have limited performance resources or are in others ways not capable of costly computations. The processing of observed data works is done similar to event correlation systems [9], but generally uses less functions and expression power. Instead the focus is on a visual language that enables non-programmers to specify observation according to their information requirements.

3 Product usage observation

Our approach separates the concerns of (i) product or system development and (ii) the specification of what to observe and how to present the collected data. In its application, product observation [5] involves accordingly two roles: the first role is a system developer, concerned with the integration of the observation module into the product. The second role, the information stakeholder, specifies observation in an easy and straight-forward process. For information stakeholders the proposed approach opens a dedicated information channel which provides potentially high quality data. Even more importantly, the observation behavior can be adapted to changing information needs remotely.

In cases of large observation systems or the integration of observation into a number of different products, the developer role mentioned above can be split up into two dedicated roles: the application developer and the platform developer. While the application developer is concerned with the integration of observation callbacks, hooks, into the product, the platform developer focuses on the observation system that is finally integrated into the product.

An observation system consists of three main layers: the authoring and analysis layer, the management and repository layer, and the observation layer (cf. Figure 1). On the first layer, it is specified what to observe and how to process and present the collected information. The management and repository layer plays mainly the role of a middleware between specification of observation and observation itself; it transports observation specifications towards the observation layer and the observed data from products back to the authoring and analysis layer. The observation layer is the place where the actual observation takes place inside the products. From the development perspective, two interesting things happen here: first, hooks have to be integrated into the product. Those hooks represent places that can be observed, that is, they are proxies to the actual places in hardware and software where the actual data is generated. This encapsulation helps to maintain a consistent interface from product to observation module. Second, observation specifications coming from the authoring layer are transformed into executable runtime structures and represent the logic of observation in a certain scenario. The latter aspect is addressed in [4].

4 Modeling observation

Modeling of observation systems is done by means of the Unified Modeling Language (UML). This language is an industry-wide standard for modeling of hardware and software systems. UML models are widely understood by developers in the community, and the modeling benefits from extensive tool support. UML offers a light-weight extension mechanism, profiles, that is suitable for building domain-specific UML models. This means to project domain language semantics onto UML by technically extending it with a dedicated set
Figure 1. Technical observation system overview

Figure 2. Observation profile (package view)
of new concepts.

A profile denotes not only a list of concepts that are to be used to build a certain kind of system or system part. Likewise, the use of concepts shall be shown. At the same time, a profile is also a generalized description of possible systems. Since the variability in the domain of observation system is high, ranging from enterprise level systems with huge performance resources to embedded systems that barely can afford the necessary processor cycles for pre-processing of collected data. The level of detail that is shown in the observation profile differs much. Parts like the interface between host system and observation component can be described in high detail whereas large parts of the data collection support system and the whole authoring and analysis layer cannot be specified in general terms. Still, in these parts, the main concepts are provided and linked in order to find a compromise between concreteness and flexibility. Sub-profiles are shown here in an overview; most contain more elements that structure the big building blocks explained here.

The observation profile as shown in Figure 2 is basically divided into five sub-profiles that can be mapped to three layers of an observation system (cf. Section 3). While the observation authoring and the observation presentation profile packages are entirely concerned with the authoring and analysis layer, the management and repository profile packages respectively belong to the management and repository layer. The remaining execution profile package is concerned with the observation layer.

In the following, the five main sub packages and their contents shall be described in detail, beginning with the authoring profile package.

5 Observation Authoring

The authoring profile package (cf. Figure 3) is divided into five sub packages: Besides the specification formalization and the authoring environment packages which describe the tooling for observation definition, the simulation package serves as a basis for the testing of observation specifications and the semantics package provides means to define semantic concepts which can be attached to specification elements. Finally, the visualization formalization is part of this sub package since it is crucial to define metrics together with their presentation in the authoring phase within one environment in order to simplify the domain experts’ tasks and leverage tools without too many context changes.

5.1 Specification Formalization

Since observations always follow a specific purpose, it is often necessary that observation specifications grasp the observation-related semantics of a certain domain. Therefore we do not propose a single language for all kinds of observation scenarios and applications domains, but instead the essential building blocks for domain-specific observation languages. The observation specification formalization sub-profile contains those building blocks. The main element is the «Observation Specification» which serves as a container for «Specification Element»s. These elements can be either «Specification Block» or «Connector», a notion which is is close to the event-driven paradigm in specifying observation. However, this can be mapped easily to other paradigms like rule-based or metrics-based definitions. Even for these different approaches towards observation specification, the sub types of blocks, «Event Source», «Processing», «Concept», and «Export» are valid. These concepts are mandatory for the development and execution of an observation system. Finally, «Syntax Constraint»s are needed to define the exact syntax of the intended domain-specific language. Both «Specification Block» and «Connector» have constraints attached that serve as a basic grammar of an observation description. However, more sophisticated semantic constraints are posed by the editor as denoted in the authoring environment sub-profile.

5.2 Authoring Environment

Inside an authoring environment, several editors might play a role for a concise definition of observation, however, only two of them shall be described here. Others, such as editors for surveys and semantic concepts and ontologies are beyond the scope of this report.

The logical first step in the observation flow is to define what should be observed and how the collected data should be processed. Therefore a comprehensive editor, «Specification Editor», is needed that provides observation building blocks in a directly manageable way. The editor provides means to construct specifications by using visual or textual «Specification Element Representation». Ways to manipulate the representations are abstracted as «Manipulation». Underlying details of this concept are auxiliary items that are beyond the scope of the observation profile. The editor has a strong dependency on the «Observation Specification Formalization» package, as that defines the actual syntax of the observation specification language, being which elements to include and to allow in connection with other elements and which constraints to
satisfy for a valid specification. That is, the editor incorporates validation of newly created observation specifications and thus ensures that all specifications comply to the syntax and other additional constraints of the domain-specific language.

The second editor is used to define an observation specific platform model of the host system. This model defines the exact properties of hooks in the observation system that can be accessed. Similar concepts are used for the definition of the editor compared to the specification editor. However, this editor results in a hook model that is a description of the interface between observation system and host system. This is the view an observer gains of the host system. Naturally, this editor has a dependency relationship to the observation integration package, more specifically, the «hook» concept.

5.3 Simulator

Before distributing an observation specification to potentially hundreds of test machines, it is advisable to test or simulate it within the authoring environment. Moreover, this allows to briefly check if the data is collected, processed and finally presented in the right way. The Observation Simulator package contains two concepts that connect to observation integration concepts and basically schedule and generate synthetic hook data which is then processed as specified previously. The «SimulationScheduler» takes in an observation specification and executes it using the «ObserveeSimulator». The latter concept denotes a subsystem which generates hook data and triggers hooks accordingly, by using either a user interface that allows authoring environment user to trigger hooks with user-defined data, or by incorporating hook trigger probabilities defined a-priori. These facilities can be used to check the routing of observation events and the processing semantics, leading to a specification that collects the relevant data accurately.

5.4 Semantics

The semantics package aims at the information level of observed data, that is, the connection of atomic events to semantic contexts. By linking simple events together, insight into more abstract usage patterns can be gained. Basically, a «SemanticConcept» is stored within an «Ontology». Presumably, multiple ontologies can be used in the specification process to divide the semantic properties of host system, experiment-related information, and other semantic content into
respective ontologies. The linking between specification and semantics is expressed with a link between the ontology-based concept and the «Concept» specification block (cf. Figure 3).

5.5 Visualization Formalization

While the main focus of the authoring profile package is on specification of observation, the visualization specification is also part of it. The authoring phase is the observation flow is naturally also the place for thinking about how the collected information is further used and presented. The visualization sub-profile targets the definition of metrics within the authoring environment that translate raw or preprocessed observation data into accessible information. These metrics can be leveraged within the analysis phase in the form of charts, data views and other visual representations. The «Metric» concept denotes this idea. A metric is based on one or more axes which represent a data filter. The «Axis» concept is abstract and sub-concepts like «AggregateAxis», «SemanticAxis», and «TimeAxis» realize different filter domains. For instance, the metric "average time for all users using features A, B, or C over every single experiment day" uses three axes: a time axis for the all days of the experiment, a semantic axis for features A, B, C, and an aggregate axis for the aggregate sum of time that the users have used the respective feature per day. This metric can be easily visualized, for instance, by means of a line chart. In the following section the concepts using such metrics are explained.

6 Observation Presentation
- Visualizer

The visualization of observation data is only one possibility to further use captured data; another is to export the data, for instance, to specialized analysis tools. Besides offline analysis techniques, the real-time visualization of collected data in the instant it reaches the server is crucial to oversee the study, to make sure all necessary data is collected, and to grasp a quick overview on data aggregates.

The presentation profile package shown in Figure 4 contains at this point only the presentation sub-profile which basically contains «View» sub-concepts. The «ChartView» is connected to a «Metric» concept and visualizes collected according to the axes defined in this metric. An «ActivityView» is specialized view for activity and status information of product instances («MachineStatusView»), but also regarding the distribution status of observation specifications. Since observation modules inside the products “pull” the specifications, a «SpecificationDistributionView» helps to keep track of the module updates. Finally, there are views that show the raw collected data. The «DataView» concept denotes such views which provide richer information to advanced users. Compared to charts which show mainly an aggregate simplification of all collected data, the data views display task data fields, semantic properties as well as originator information.

7 Observation Management
- Specification Distribution

Before observation specifications can be executed on the product instances, they have to get there first. The observation system provides a middleware that bridges the infrastructure gap between authoring environment and product instance. The specification distribution sub-profile (cf. Figure 5) denotes the necessary concepts for such a distribution middleware. Commonly realized as a server, it provides two interfaces: the «SpecificationPublisher» concept which permits the authoring environment client to send a specification to the server, and the «SpecificationSubscriber» that provides access to new specifications for the observation module client, in this context called a «Machine». As soon as the latter interface is accessed by such a machine, the «ObservationDistribution» compiles an «ObservationPlan» that contains all valid and active «ObservationSpecification>s for this machine. The machines analyze the received plan and update themselves with new specifications in case of a changed plan. The subscriber interface contributes events to the activity and update status of certain machines, too.
8 Observation Repository

The repository layer of an observation system is responsible for collecting pre-processed data from local observation units (directly) and from proxy units that bridge a potential technical gap between local units and repository server. The two main parts are the data storage sub-profile and the data access sub-profile (cf. Figure 6). While the first serves as a general data storage place for all sources of information, the latter handles access for the observation-integrated analysis components and external tools that require unfiltered data access.

8.1 Data Storage

The storage of observation data is technically one of the most demanding tasks in the whole system as potentially a large mass of data items can be collected. However, the description of observation-related parts is rather short since flexibility regarding the data base and data base abstraction is crucial. «DataCollector» is an interface similar to the ones in the specification distribution sub-package. It allows the observation module client to access the data collection server and send collected data which is subsequently stored according to a «DataSchema» in the data base. As mentioned before, the data base access has to be flexible, so a «DatabaseAccessor» serves as a proxy to a data storage implementation.

8.2 Data Access

Observation data access use used by data consumers such as the internal visualization on the analysis layer and external tools that might require different data formats and support only few access methods. Basically, this sub-profile shown in Figure 6 provides a «DataView» that contains a «Query» to access a «DataSource». This source directly links to the «DatabaseAccessor» which abstracts from the database. In addition, the «DataView» involves a specialized «ExportInterface» which enables external data consumers to access the collected observation in the right way.

9 Observation Execution

The observation execution profile package is concerned with both observer and observee parts of the observation system. In its application, it is almost entirely built into the product and accesses the host system via a hook interface to acquire data. Inside this profile package there are two sub-profiles that describe architecture of observation and its integration into products. Another part deals with the observation behavior at runtime, and the forth package provides structures for the observation data that is eventually collected.

9.1 Integration

In the specification of observation, hooks are used as information sources. However, hooks are only abstract places where information can be perceived. Therefore, on the system level, a «Hook» is realized as a proxy
element that encapsulates the combination of an «Observable» element and its observation-related properties, such as «Characteristic» and «Constraint». Characteristics cover timing properties and data types, constraints describe runtime limitations of the observable. This meta-information can be used to build predictable observation modules, or to simulate observation behavior prior to implementation or deployment.

Hooks and observables are basically two different views on the same entity. From the hook side, only observation-related properties are shown and other information, e.g. about the implementation, is hidden - vice versa from the observable side. Both stereotypes are aggregated in respective stereotypes, «HookModel» and «Observee». While the hook model is simply a collection of hooks without further meaning, the observee denotes a system part which contains «Observable» elements, but is itself not directly observable. This stereotype can be used early in the development to annotate system parts that should be observed, and can be refined later to actual «Observable»s. Another stereotype of the integration profile is the «ObservationContext» which represents contextual information belonging to observable or observee. This information can (i) determine how the observable behaves, generates data, and responds to triggering, and (ii) it can be part of the raw observation data that is generated by the observable. All context information depends on the «ObservationScenario». Such a scenario is a usage setting, e.g. an experiment setup, and contains information about the environment the product is used in, as well as the user who interacts with the product.

To further explain the relationship between hooks and observables, we will have a look at interaction patterns in the observation domain. The nature of hooks, being either self-triggering, externally triggered, or both, suggests basically two interaction patterns. The self triggering and the externally triggered patterns are explained in Figure 8 by using the aforementioned stereotypes of «Observable» and «Hook».

The first pattern is suitable for hooks which are self-triggered, that is, the observable system structure autonomously triggers the respective hook object whenever new information is perceived and should be fed into the observation system. In this pattern the responsibility of taking action lies on the observee’s side.

The second pattern deals with hooks that have to be triggered externally in order to produce data. In this pattern, the hook object is linked to the observable structure, e.g. in the form of a public operation, and can trigger the observable. In the rare case that an observable has both characteristics, a combination of those patterns is also possible.

Hooks, as their proxy nature suggests, connect observables to the respective interface in the observation module, the «HookInterface» as shown in Figure 7. The observable element delivers raw data to the interface, and inside the module this interface presents the data to the observation component. The component subsequently processes the raw data according to the specified observation behavior. Obviously, the resulting data is determined to a large extent by the observation system input coming from hooks, thus the strong connection to the «HookData» stereotype (cf. Figure 7).

9.2 Architecture

The «HookInterface» is one of the main parts of the observation architecture profile shown in Figure 9. The interface stereotype is a part of the «ObservationModule», namely being a sub stereotype of «ObservationSubModule». Other sub modules are concerned with the communication between observation and repository.
Also, an observation module could possibly have a user interface in order to inform the user about the observation or to collect subjective feedback in the form of questionnaires.

Two submodules deal with runtime behavior of specified observation, the observation component. The «Configurator» receives an observation specification and translates it into an executable «ObservationComponent» which is run by the «Scheduler». The latter module is responsible for triggering of hooks and the synchronization of concurrent events.

### 9.3 Behavior

The behavior sub-profile (cf. Figure 10) describes the dynamic runtime structures that form an «ObservationSpecification», and that are executed within the observation module inside the host system. These building blocks are dynamically created and linked, and enable a flexible adaptation of observation logic to changed requirements. Corresponding elements are denoted in an observation specification which is a model of the observation behavior that is subsequently created and executed. The structure of the behavior sub-profile is in this sense similar to the specification formalization sub-profile structure.

The basic functionality a dynamic building block needs is expressed in the «FlowNode» concept: event routing behavior. «FlowNode»s can be connected via «FlowRoute»s. While incoming routes of a node are independent and can even be assigned a name for a parameter, outgoing routes are all triggered once the node fires an event.

Concrete realizations of the abstract «FlowNode» concept are: (1) «ProcessingNode»s which are capable of processing the data contained in incoming events, (2) «ExportNode»s which cache, serialize and transmit incoming events to the «Communicator» (cf. Figure 9), (3) «SemanticNode»s which attach semantic concepts to an incoming and instantly outgoing event, and (4) «Hook»s which are the entry points for data and events in the «ObservationComponent».

According to different uses of the hook, there are several types of hooks: «PlatformHook»s which connect to the host system (that is, the «Hook» concept) and acquire data therefrom. «SystemHook»s capture events generated inside the observation system, being for instance status messages and error codes. Finally, there are «SemanticHook»s which capture semantic events. These events are triggered whenever a semantic concept is attached to an event in the «ObservationComponent». Especially this enables information stakeholders to quickly abstract from hooks and low-level events by means of user-defined semantic concepts. Once these concepts are bound to events and data, a considerably easier specification of observation logic can be achieved.

### 9.4 Data

Being almost intangible within the observation system, but becoming the most stable artifact of observation once it reaches the data collection server, the structure and properties of observation data are denoted in the data sub-profile (cf. Figure 11). When it is generated or captured in its raw form in a «Hook», the «HookData» can have two modalities «HookEventData» and «HookStatusData», the former expressing an event which occurs at a distinct time point, the latter expressing a sample taken from a continuous data stream. As soon as this data is processed in the «ObservationComponent» it becomes «ComplexEventData» connected to an «Originator» and potentially «SemanticConcept»s. However, all types of «ObservationData» can be exported and lead to a valid data trace.
10 Conclusion & Future work

Companies experience a lack of reliable usage information about their products. Our approach towards this problem is to build observation modules into products that are capable of providing reliable and structured product usage information directly from the source. Observation integration can possibly have a strong impact on the development of innovative products, thus the need for doing this efficiently in an engineering methodology. This report introduces a model-driven technique to integrate observation functionality into products. The profile introduced here simplifies the development tasks necessary for observation integration, thus reducing the effort for integration. It helps to automate the process of observation specification and data collection. However, the issue of automation remains partly future work as the provision of even better tools for the observation integration is crucial. Furthermore, we see observation integration not as a simple parallel development task only, but as a potential driving force behind a new development paradigm: “design for observation”. This involves observation as a first class development aspect and helps to provide a solid basis for extensive, but meaningful product information presented to information stakeholders in a comprehensive way.

Acknowledgments

This work is being carried out as part of the “Managing Soft-Reliability in Strongly Innovative Product Creation Processes” project, sponsored by the Dutch Ministry of Economic Affairs under the IOP-IPCR program.

References


