Model Interpretation for Executable Observation Specifications

Mathias Funk, Piet van der Putten, Henk Corporaal
Dept. of Electrical Engineering, Electronic Systems Group
Eindhoven University of Technology, The Netherlands

E-mail: {m.funk, p.h.a.v.d.putten, h.corporaal}@tue.nl

Abstract

Observation functionality integrated into interactive products can help companies identifying current consumer requirements and expectations. As these needs can change rapidly, detailed information about product usage that comes from habitual interaction is crucial to evaluate product acceptance and relevance. We explore how products can be extended with observation functionality that satisfies the information needs of multi-disciplinary experts in the development team. In the process of product evaluation, information requirements are bound to change, and so is the observation behavior. Our approach addresses this by integrating observation functionality into products which can be adapted to current information needs. This paper presents a novel way to remotely configure products in the field by using high-level models, graphical observation specifications, that are interpreted by a runtime environment built into the products in the field. An industrial case-study shows the applicability of the approach. This work is part of ongoing development aiming at a generic observation integration methodology.

1. Introduction

Complex consumer electronics products as well as other innovative product categories nowadays integrate many different features in order to serve a large group of customers. The mass of functions has to be accessed through a user interface which in turn gets more and more complicated. Users have problems to find their ways. Increasing numbers of returned products without any detectable failures suggest this [3].

Furthermore nowadays product creation processes are characterized by high complexity of products and they are influenced by rapidly changing customer demands. Hence, an up-front specification of the product becomes hard if not impossible. In the past, products could be improved in the next version, but today the markets often demand completely new products. Technologies have to reach a level of maturity in a shorter time. The lack of information about the customers' needs leads to a situation where companies press functionality into products, thus entering a vicious cycle of complexity [1]. This blurs the customer's understanding of the product and a match between customer expectations and the actual capabilities of the product becomes even more unlikely.

An approach to address this industry-wide problem is to get representative user feedback on try-out products or prototypes [2]. Traditionally this is done by collecting customer opinions in questionnaires and video-taping user interactions with the product in usability labs. Nowadays, with almost ubiquitous internet access, other methods can be used which are expected to provide much richer data on the actual used product features and user preferences. The integration of observation modules into products can enable data collection according to the actual and ever changing information needs of the product development team.

Our research aims at the introduction of observation integration or *design for observation* as a first class development task, because the delivery of relevant information of use and possibly user expecations will become more important in the future of product development.

In this work we address a problem that occurs when the development processes are not yet tuned to observation integration in an efficient way: observation is brought into products late in the development process. Due to changing requirements for observed information, the implemented observation functions have to be adapted regularly. This holds especially for the use phase, when products are given to testers. Then, highly adaptable and remote observation is crucial. This causes a substancial effort not only for developers but also for observation specification and certainly for the alignment of both. If the adaptation mechanisms are not automated, product evaluation becomes difficult if not impossible. Therefore automation of observation specification deployment is a primary precondition for such product evaluation methods.

More precisely, we address the technical transition process between observation specification and the dynamic execution of a translated specification, possibly to be constructed during runtime. As a result, observation facilities support this scenario of remotely adapted observation via a communication channel like the internet. Figure 1 shows an overview on such a system, consisting of an authoring environment, a server instance, several products in use and further services dealing with the analysis and visualization of collected product usage information. We refer to [8] for an explanation of system and its use to model users and their interaction with a consumer electronics product.

In the following sections, after pointing at related work, the approach of observation modeling is explained together with a description of visual specification and observation modules. This is followed by the core section about model interpretation. The paper ends by presenting a case study which shows the applicability of the approach in the context of new product development.

2. Related work

The research on observable products as described in this paper is on the one hand strongly connected to the field of user modeling. The framework we developed stands in the tradition of generic user modeling systems (for an extensive overview see [12]). On the other hand, it is in the area of remote runtime monitoring where there have been efforts to monitor deployed software [4] and to use logging inside products [13]. The use of a client-server architecture for information distribution across a network of products is straight-forward in this domain and has been described before [10]. However, our approach of remotely changeable observation behavior differs from traditional monitoring as we do not assume a pre-defined set of information sources, but deal with constantly changing information requirements. In this sense, the research stands also in relation to adaptive software [11]. This paper tackles the problem of *flexible instrumentation of observation modules*. Our approach is based on the specification of observation by use of a domain-specific language [6]. The specified observation is executed on products, which is an application of model interpretation [5]. Compared to well-known model-driven approaches like MDA and MDE [14, 7] this technique offers a dynamic transformation shortcut from model to executable.

3. Observation

The observation of remote systems potentially covers a wide range of complex electronic products. It is seldomly done in an engineering approach aiming at reuse and a long-term application. Especially for product families observation gains importance: it is one of the few system parts which are easiest to generalize. Moreover, the collected information has a large influence on the specification and the targeting of future products within the product family.

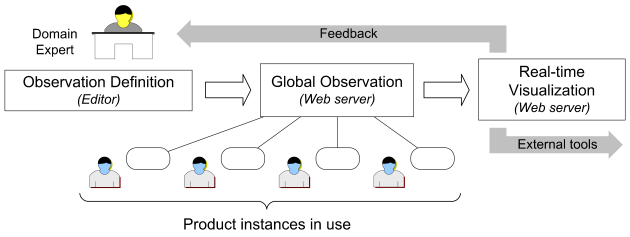

Figure 1. Framework overview

Observation is done in several subsequent steps: Information is sensed by so called *hooks* which might imply that an information source is either triggered periodically for data or raises an event itself. Resulting low-level data is processed in the next step and can herewith be aggregated, normalized or temporarily cached. This preprocessing stage yields complex events which result from the combination of multiple low-level sources. Depending on the extent of aggregation and event correlation those events can carry enough semantic information to be relevant for analysis by information stakeholders. Finally, the data has to be collected centrally which allows for real-time visualization and post-processing using external tools. Obviously, information capturing and preprocessing which are performed on the individual product instances have a huge impact on the quality of the information that is presented to post-processing and analysis.

3.1. Observation system

An observation system (cf. fig. 2) consists of three main layers, (i) the *authoring and analysis layer* where specification of observation and the captured information is worked with, (ii) the *repository layer* which accomplishes the task of configuration and data aggregation, and (iii) the *observation layer* with local product instances. Observation specifications have to transmitted to product instances and observed information has to be captured in a central instance for further analysis. In the optimal case, a knowledge engineer can define an observation specification and the infrastructure provides all services necessary to configure product instances and transport the data back for analysis.

In this and following sections, we will concentrate both on a part of the authoring and analysis layer and the observation layer. The aforementioned specification of observation consisting of (i) hooks, (ii) processing, and (iii) export, can be modeled by a visual language using few

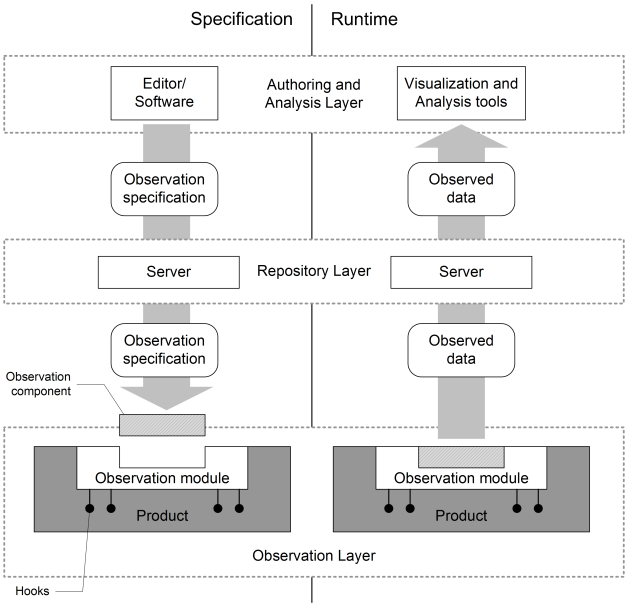

Figure 2. Observation system overview

graphical building blocks (see section 3.2). On the observation layer a runtime structure called *observation component* (OC) is constructed from a visual observation specification by means of model interpretation. An OC is a pluggable part of the observation module that is connected to a specialized runtime environment inside the observation module (cf. fig. 2).

The development effort for a working observation system as a whole can be split up into two main tasks: (i) integration of observation into systems including a middleware capable of information delivery between the observation layer and the repository layer, and (ii) the definition of observation via the visual language. This separation of concerns supports the roles involved in the process: *product developers* handle the first task and *information stakeholders* define what should be observed. Ideally, a third role comes in, the observation developer, who shoulders the burden of observation-specific programming which includes, for instance, the infrastructure, editor customizations and platform-specific adaptations of the observation module.

3.2. Visual language

Observation specification should be performed by experts in the domain of user-related information or other product information stakeholders. Often those people do not have the necessary system engineering and programming skills to instruct a distributed system of product instances. Therefore we propose a visual specification language that hides low-level programming matter and enables domain experts to take advantage of their special knowledge about product information. It is a domain-specific language that focusses on observation only. Likewise, concepts of general purpose programming languages which are inappropriate for the specification can be left out. The essential language elements shall be described in the following.

Hooks are places for information retrieval and they are basically the only information inlets of the observation system. There are two types of hooks, the ones that have to be triggered to yield data, and the ones which trigger themselves and can be seen as manifestations of events in the over-all system. As event generators, hooks are also the sole platform-specific parts of the system and represent an interface between the product's internals and the observation module. The hooks that are not self-triggering can be linked to *timers* which simply realize periodic signals that cause those hooks to fire. This sampling technique is used especially for information like performance measurements or resource load that has a continuous nature.

Hooks generate low-level system data that is mainly not immediately useful for analysis. It is a mass of atomic system events, that has no inherent structure and does not gain any comprehensive results - let alone answering specific questions. Therefore this data has to be preprocessed to become meaningful. The next stage of the observation specification tackles this aspect. Hook data is routed through *processing blocks*. Processing can involve calculations to normalize incoming numbers or the correlation of multiple events to gain derivative complex events. Closely related are *caching blocks* that enable data snapshots and can, for instance, be used as a sliding window over an event stream to compute a floating average.

After finishing those early computations the information shall be exported. This means to transfer it from the product instance to the repository layer that gathers all product usage data in a central data storage. For this purpose the visual language contains an outlet symbol, which can be used to route outgoing information and to label the information according to the semantics it represents.

All aforementioned visual blocks can be linked by *routes* which connect the outlet of a block with inlets of other blocks, thus forming a directed graph (cf. fig. 4 for an example). An observation specification denotes an event-driven system that reacts on the occurence of events and may also trigger hooks by the use of timers. Still, from the user perspective the flow of information in such a description can be seen relatively easily and the language abstracts from concurrency issues as well as from potential data conversion problems. Its visual form allows to concentrate on the matter of specification on the information level.

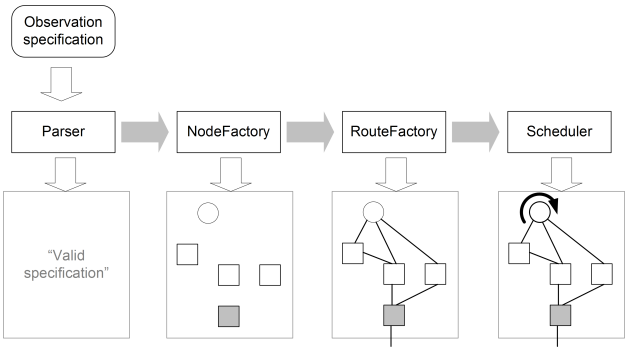

Figure 3. From observation specification to observation component

3.3. Observation module

The visual language's counterpart on the observation layer (cf. figure 2) is a module integrated into the system to be observed. The degree of integration into the host system depends on the required access on information sources. An observation module can be realized in multiple fashions, e.g. as a plugin for existing software, as separate software or even as a dedicated hardware subsystem - as long as the basic requirements (i) access to information sources and (ii) communication capabilities with a repository layer are fulfilled. For less strict requirements, observation modules can accompany a system as long as its services are needed and should be easy to remove after use. Also the impact on system performance and of course privacy as well as security issues have to be considered carefully, but are out of scope here.

As mentioned before, the module provides internal execution capabilities: it receives an observation specification, constructs and runs an OC, and delivers the collected information towards the repository layer. Therefore it consists of several parts that play a role in communication, configuration and the observation itself. The communication subsystem jointly realizes the data transmission infrastructure depicted in figure 2 together with a server on the repository layer. The configuration subsystem essentially contains the parts shown in figure 3 which parse and construct an OC from a specification (further discussed in 4.1). The building blocks of an OC are particularly interesting in the context of the next section and shall be described there.

4. Model interpretation

Model-driven engineering being one of the most influencal achievements in recent system development is also at the core of technical observation development. Especially the agile nature of observation development and iterative characteristics of the process require an automated flow so that changes in observation requirements can be propagated quickly towards actual execution [11]. Furthermore, it is crucial to protect the client machines from potentially harmful virtual machine bytecode or, potentially worse, binary code. Still, the highly dynamic nature of the product evaluation settings demand an special engineering approach: runtime structures are constructed directly from the specification. This technique replaces the transformation and code generation steps of traditional MDE with a single interpretation step. Code generation in principle transforms a model into a textual representation which is processable by, e.g. a compiler. In contrast, model interpretation directly processes the model and generates executable structures in memory. This has the main advantage that the model can be *embedded* into the runtime system. This emphasizes the safety of the system and allows for a change of the observation behavior at runtime, simply by replacing the interpreted model with a newer version.

4.1. Observation specification

For the specification of observation we developed an editor based on the Eclipse platform. That editor allows for an easy composition of an observation specification suitable for domain experts. Also, it offers the possibility to send the finished specification directly to a server on the repository layer which is part of the distribution infrastructure for updating product instances. Figure 4 shows an example of a graphical observation specification. It denotes the timed triggering of a hook requesting information about the CPU performance every ten seconds. This information is averaged (*avg* symbol) by a processing node and exported via an export node.

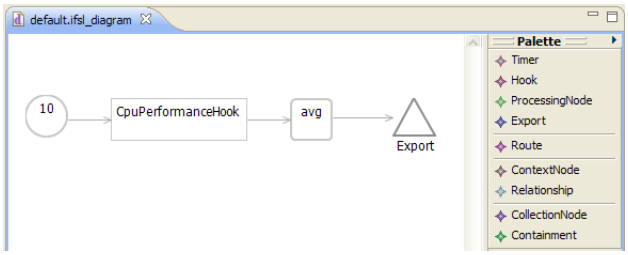

Figure 4. Visual editor screenshot with an example specification

Models expressed in the visual language are serialized in plain XML. This can be parsed by the observation module. Corresponding to the example specification shown in figure 4, its structure also appears in the XML file that is sent to the observation module for execution (cf. figure 5).

```
<?xml version="1.0" encoding="UTF-8"?>
<ifsl:Model xmi:version="2.0">
  <elements type="ifsl:Timer" period="10"
      unit="seconds"/>
  <elements type="ifsl:PlatformHook" name="
      CpuPerformanceHook"/>
  <elements type="ifsl:Route" end1="//
      @elements.0" end2="//@elements.1"/>
  <elements type="ifsl:ProcessingNode" name
      ="avg"/>
  <elements type="ifsl:Route" end1="//
      @elements.1" end2="//@elements.3"/>
  <elements type="ifsl:XMLExportNode" name=
      "Export"/>
  <elements type="ifsl:Route" end1="//
      @elements.3" end2="//@elements.5"/>
</ifsl:Model>
```

Figure 5. XML version of observation specification

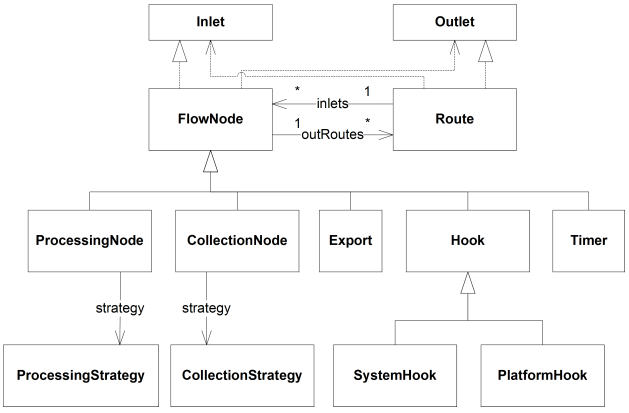

Figure 6. UML class diagram of observation component building blocks

4.2. Observation runtime

The observation module contains a runtime environment that can execute an OC as specified by the visual model. Figure 3 shows the configuration process. An observation specification is parsed and checked for validity. After that, a set of building blocks is constructed dynamically using a *NodeFactory*. The routing unit takes this set of blocks as input and creates routes according to the specification. Implicitly the export nodes of the network are connected to the respective communication facility. Finally, the scheduler subsystem starts all timers, the only active parts in an otherwise reactive event-based architecture.

To construct such a network, the event-driven executable system makes use of the base class *FlowNode* which realizes the basic routing functionality together with the *Route* class. As the UML diagram (see figure 6) shows, dynamic linking of nodes is accomplished by using the *FlowNode-Route-Inlet-Outlet* pattern: *FlowNodes* offer inlet functionality by means of a provided interface and a *Route* can connect to those nodes with a usage relationship with the interface *Inlet*. For the interface *Outlet* the reversed relationships hold. Objects are linked together by means of the *inlets* and *outRoutes* associations.

The UML class diagram in figure 6 depicts also the classes that represent hooks, processing nodes, collection nodes, exports and timers. Obviously there is a 1:1 relationship between elements of the visual language and the instanciated objects that are linked together by means of the inherent routing functionality of all objects derived from the base class. What the picture also shows is the application of the strategy pattern [9]. It helps to realize different kinds of behaviour of *ProcessingNodes* and *CollectionNodes*. Depending on the type of processing or collection specified, different strategies can be chosen. This especially reduces the effort for observation development as only necessary computations have to be implemented in the observation module.

Yet, the implementation of hooks proved to be the most demanding task of observation development as it means to interface the product at various levels, a task that strongly depends on documentation and openness of the platform.

There are basically two types of hooks: Platform hooks which access the host system and are characterized by mainly platform-specific behaviour, thus the name, and system hooks that access the observation module. While the use of platform hooks for information collection is straightforward, system hooks capture events concerning the observation itself. In the future, this can be used to adapt the observation to context changes or to establish semantic links between observed items on multiple levels.

5. Case study

Together with a large Dutch electronics company we carried out a case-study to test the observation of a consumer electronics prototype. This showed the applicability of the approach in a world-wide observation scenario that connected 20 machines spread over 8 countries to a central server which collected about 800.000 data items. The product instances were pre-configured before roll-out. As expected, changes in the observation requirements of information stakeholders demanded for remote changes of the observation specification which were performed successfully several times. The machines continued to capture data according to a new observation specification.

The data collected during the case-study supported mostly the assumptions of the development team about product usage, but also gave new insight on country-specific usage problems and customer expectations. Regarding the successful application of the proposed technology and new insight into product usage it has been decided to continue with successive experiments on a later version of the observed product prototype.

6. Conclusion & Future work

Regarding the fact that the majority of products are currently not designed for observation, we are working in the direction of a design method for self-observing systems. To provide an intermediate solution for observation integration we chose model interpretation for maximum flexibility and agility. Our approach is to specify visually and to execute the finished specification directly on the product. This emphasizes the separation of concerns between domain experts who are interested in the collection of usage information and developers who are concerned with the system engineering.

We developed an experimental framework for specification and implementation of observation functionality. A new visual specification language has been introduced to support domain experts specifying observation behavior. It proved to greatly simplify the task of product usage data collection. The language is generic enough to be reused in different observation contexts. Only minor changes have to be made to the observation specification runtime environment in case a new product software implementation platform has to be entered.

The case-study showed that as soon as observation modules are in place and the specification supports basic measurements of users' interactions with a product the need for better semantic linking between observed data arises. So far, a lot of effort still has to be spent on the post-processing of captured data. The next step is an annotation of events with semantic information that tells e.g. about the origin, conditions and context of such an event. Also this is done in a structured way, such that the information can be easily exploited during post-processing using automatic analysis tools. Another future direction is the collection and incorporation of subjective user feedback data into the set of objective product data. In addition, subjective data can help to understand the *why* in user-product interaction. It will be possible to bind subjective feedback measures to the occurence of certain events which enables a dynamic insight into the usage process together with background information coming directly from the user at the same time.

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

[1] S. Bly, B. Schilit, D. W. McDonald, B. Rosario, and Y. Saint-Hilaire. Broken expectations in the digital home. In *CHI '06: CHI '06 extended abstracts on Human factors in computing systems*, pages 568–573, New York, NY, USA, 2006. ACM Press.

[2] R. Cooper and E. Kleinschmidt. New products: What separates winners from losers? *Journal of Product Innovation Management*, 4, September 1987.

[3] E. den Ouden, L. Yuan, P. J. M. Sonnemans, and A. C. Brombacher. Quality and reliability problems from a consumer's perspective: an increasing problem overlooked by businesses? *Quality and Reliability Engineering International*, 22(7):821–838, 2006.

[4] M. Diep. Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Trans. Softw. Eng.*, 31(4):312–327, 2005.

[5] J. Estublier and G. Vega. Reuse and variability in large software applications. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 316–325, New York, NY, USA, 2005. ACM.

[6] E. Evans. *Domain Driven Design*. Addison-Wesley, 2004.

[7] D. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[8] M. Funk, P. van der Putten, and H. Corporaal. Specification for user modeling with self-observing systems. In *Proceedings of the First International Conference on Advances in Computer-Human Interaction*, 2008.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[10] K. Kabitzsch and V. Vasyutynskyy. Architecture and data model for monitoring of distributed automation systems. In *1st IFAC Symposium on Telematics Applications In Automation and Robotics*, Helsinki, 2004.

[11] J. Karsai, G.; Sztipanovits. A model-based approach to self-adaptive software. *Intelligent Systems and Their Applications, IEEE [see also IEEE Intelligent Systems]*, 14(3):46–53, May/Jun 1999.

[12] A. Kobsa. Generic user modeling systems. *User Modeling and User-Adapted Interaction*, 11(1):49–63, Mar. 2001.

[13] J. Kort and H. de Poot. Usage analysis: combining logging and qualitative methods. In *CHI '05: CHI '05 extended abstracts on Human factors in computing systems*, pages 2121–2122, New York, NY, USA, 2005. ACM Press.

[14] B. Selic. The pragmatics of model-driven development. *Software, IEEE*, 20(5):19–25, 2003.