Specification for User Modeling with Self-observing Systems

Mathias Funk, Piet van der Putten, Henk Corporaal
Dept. of Electrical Engineering, Electronic Systems Group
Technical University Eindhoven, The Netherlands
Email: [m.funk, p.h.a.v.d.putten, h.corporaal]@tue.nl

Abstract

The complicated user interfaces and complex functionality of nowadays interactive products lead to a new class of failures: People do not understand their products and thus fail to use them successfully; many products are returned for which no detectable errors can be found. These field problems of interactive products cannot be found by traditional testing methods. Industry needs reliable and structured information about the users' behavior to get understanding about the root cause of so called soft product failures. In this paper we present a framework that helps usability and quality experts to derive user models from product observation. This is supported by a novel visual language for specification what should be observed and how collected data is represented, and a system architecture for distributed self-observing systems. This approach separates the concern definition of observation from the implementation of observation facilities.

1. Introduction

In the context of nowadays highly intricate consumer products, e.g. innovative digital products or professional office products, there is often a mismatch between expectations of customers and the specification of products. Complex products with complex user interfaces offer a lot of different functions which results in a large mental effort to "learn" a product. To complicate this even more, current interactive products implement multiple ways to reach a goal. Customers are confused to a large extent. The tradeoff between user needs and the required effort to master the product is such that often only a minimal amount of the product's functionality is used. Customers are not satisfied, though the product works according to its specification. When contacted, service centers cannot find any faults with the product. This causes that products are returned to the shop or reside lost in boxes on the attic.

Although there exist sophisticated techniques to find (hard) failures in products for many years now, those failures cover only a decreasing share in the overall failure statistics. A new class of *soft faults* [1], [2] appears. Those faults often cannot be tracked to the product itself and partly result from customers' inabilities to use products successfully. Nonetheless, for commercial reasons industry has to approach this matter as "product faults". The more complex a product is, the more soft faults arise. Especially during usage in their habitual environment users experience product failures that are different from those discovered in usability labs or other in-house testing environments. Until recently, companies had no means to acquire information of in-the-field usage problems and had to rely on the small share of products coming back for repair or replacement. Even then, the only information that is tracked and fed back to the development team is related to the actual malfunctioning of the devices. Moreover, the gathering of information is often complicated by outsourcing the tasks of repair and replacement of malfunctioning parts to subcontractors. So, it can be stated that mostly no real structured data about the usage of products is coming back from released products.

Our approach tries to establish a novel communication channel from product to company that enables the collection of usage data by the product itself. Information about the use and performance of products in the field is fed back to the company and can be incorporated in product development and quality assurance. Figure 1 shows user modeling as integral part of an iterative product development process. In this paper we focus on product observation which results in collected usage data. This information is subsequently interpreted, a process that should finally give understanding of the user, but will at first come up with new requirements for observation. This cycle is intended to result in relevant product usage data.

People interested in usage information have not the

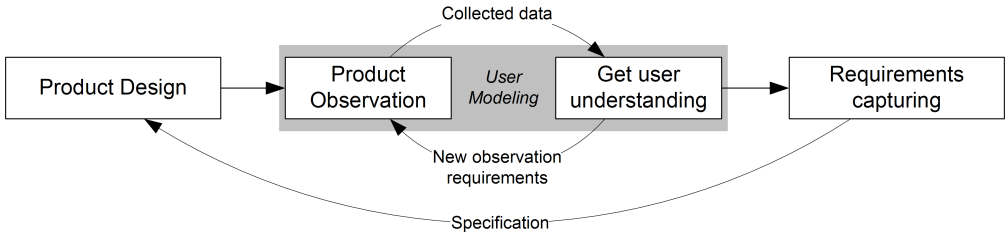

Figure 1. User modeling as a part of the product development process

same profession as system engineers or programmers; they are quality engineers, interaction designers, product managers, and service coordinators. In contrast to system developers, this special group of stakeholders has probably a much better vision on what should be observed. Consequently, the definition of items to be observed should not be done via a programming language that uses unintuitive concepts and takes unnecessary effort to learn. Instead the definition should be close to the domain of observation and should abstract from the architecture and design of self-observing products.

The solution that addresses this problem consists of two parts. First a graphical language for product-independent specification of observation behavior. Second, the engineering counterpart, there is a system-architecture which supports an easy integration of observation functionality into a product. The goal is to split the definition of observation from the technical implementation that makes a system observable.

In the following sections, after pointing at related work, our approach to modeling the user's behavior shall be described, followed by a description of a novel visual specification language for observation and an architectural overview. The paper ends with an ongoing validation of the approach by presenting a case study. The research is carried out in the context of a multi-disciplinary project together with researchers from industrial design and technology management.

2. Related work

The research presented here is strongly connected to the field of user modelling and the framework we developed stands in the tradition of generic user modelling systems (for an extensive overview see [3]). As user modeling is a heterogenous field, there have been several approaches to structure the domain [4]. User modeling and profiling are also often connected to web-based systems [5] and ambient systems [6]. In the area of remote monitoring there have been efforts to monitor deployed software [7] and to use logging inside products [8]. This work emphasizes the need of simple means for professional users that are less knowledgeable about software development to define executable systems, leading into the domain of end-user programming [9] and visual languages [10].

3. User modeling

Most users do not understand the internals of their products, but still want to achieve goals by interacting with those systems. Thus they form each their own mental model of the product. These models are structured mental pictures of a product and its interaction possibilities [11]. For complicated products, people often fail to create an appropriate mental model, but rely rather on a misleading fantasy picture, featuring false expectations that consequently can lead to customer disatisfaction. In case a user's mental model is not according to reality, it gets harder to use the product [12]. Also product designers may have an unrealistic mental model of the user [13]. The question is, can we improve the design process such that a better match can be achieved between designed product and user experience. This improvement cannot be expected unless we get better understanding of the user's interaction patterns, habits, problems and mental models.

The process of observation and interpretation (see figure 1) provides *user models* that concentrate on specific parts of the user's interaction with the product. We do not believe in all-purpose user models for product research. Models should be problem-specific and should structurally adapt to different user types and application areas [14]. The definition of observation determines the input for a user model. For instance, observation can focus on the usage of specific product parts or can involve observation of different system parts, resulting in data that pose a problem-oriented view on the interaction. Depending on the product and its environment, even contextual information can be collected. Especially in combination with user interaction information, this can give means to approach soft

faults and to find root causes of the mismatch between user expectations and the actual product.

In general the information sources, in the following called *product hooks*, are regarded as abstract places where events are recorded or values are sampled. This abstraction separates the definition of observation from the actual target system. The range of captured information can vary from rather low-level events like key-presses and application window's state changes to high-level events that imply already some semantic knowledge, for instance, about the user interaction. Still, in most cases low-level data is the sole source of substantial information. Sometimes, basic events of this kind have to be combined to more meaningful complex events [15]. Various operations might be applied, such as normalization and the correlation and aggregation of data.

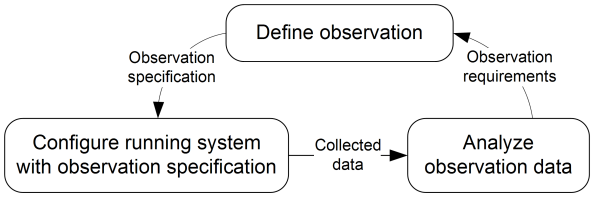

Figure 2. Observation: from definition to analysis, to definition

Also, data collection can be reactive. For example, the occurence of one event at hook A triggers an computationally expensive information retrieval from another hook B. The two data are then combined into a new complex event C. An alternative is to simply trigger hook B periodically, however it becomes clear that the first approach obviously uses less system performance. Furthermore, it also reduces the amount of collected information to most the relevant parts. This is something which cannot be achieved with the traditional approach to combine logging techniques with offline information processing. Finally, complex events are aggregated into a data structure which supports further offline processing, visualization, and interpretation.

Moreover, an important aspect is that probably observation requirements change over time, this is known as *concept drift* [16]. The gained insight of one first observation can be a trigger to more in-depth information needs, as well as separate user models for different user groups once they are classified. The proposed approach supports a fluent change of observation by updating the observation components in all local products automatically. We envision an iterative cycle between the definition of observation instruments, the actual data collection, and analysis (see figure 2). This cycle can be performed many times and can support very flexible methods of product usage research. Experiments do not have to be planned largely in advance, but can incrementally develop over time and adapt to recent findings - leading to more precise data and thus a novel picture of the user's interaction with system and environment.

4. Visual language

The key to this research is to involve the right people by building a strong connection between the stakeholders of observed information and the definition of observation. We want to provide them with tools that enable that they define themselves what information needs to be observed. The ways to do this should have a natural fit to domain and thinking of these stakeholders. We developed a visual language that is intended to have a natural expressivity for the task of observation specification. Therefore the language abstracts from engineering tasks that will be performed subsequently in order to realize a self-observing system in the whole. Instead, a few key concepts serve as abstraction layer that hides the actual concurrent, event-driven observation system. Observation can be specified using the following concepts:

- *Product hooks* represent places where actual data is measured or collected. So hooks are sources of product events that possibly also carry data samples. Hooks can deliver events on occurence, but also can be triggered.
- *Timers* trigger product hooks periodically and are adjustable to various durations, from milliseconds, minutes, hours to weeks and months.
- *Data collections* store incoming events and support lists, sets or sliding window concepts.
- *Processing* nodes compute and aggregate events or event data.
- *Routes* connect the afore mentioned objects and ensure timely data and message transfer.

The abstraction enables a straightforward definition of observation without consideration of platform characteristics, programming tasks, data transfer and storage, and automatic product configuration.

Ealier versions of the language contained various different elements that specified seldomly used aspects of observation and partly did not abstract well from technical details. For instance, we removed language elements that combined data storage with the computation of aggregates. The visual language evolved to its current state with just a few concepts (see above) that

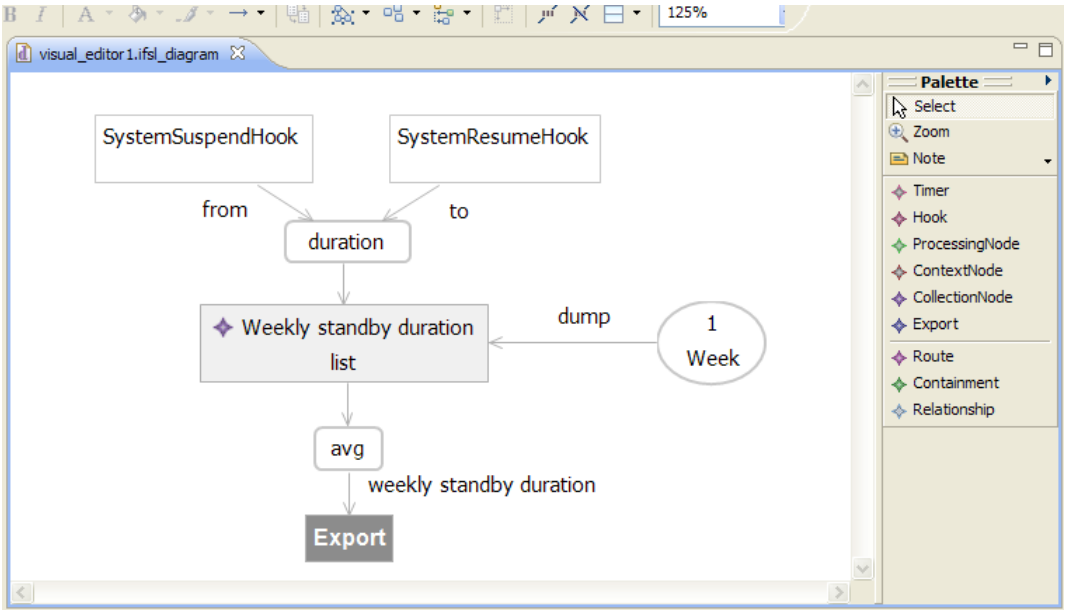

Figure 3. Visual editor with observation definition example

still cover the intended functionality. We considered this state mature enough to develop tooling.

The graphical editor depicted in figure 3 provides building blocks for the key concepts (see the palette on the right side of the figure) of the visual language. It offers graphical assistance to define the intentional structure of a self-observing system. The visual language makes products accessible in terms of observation data sources. It provides intuitive means to define a precise specification of observation, that can directly be used to actually perform observation tasks inside products.

For an educational example see figure 3. This use-case simply tracks the average duration of stand-by time, and weekly exports this information to the global observation. The stand-by time is defined as the time span between a system suspend and a subsequent system resume event. For this two hooks (*SystemSuspend*, *SystemResume*) are incorporated and trigger a processing block *duration*. The naming of ingoing routes intuitively defines how the data should be correlated; in this case the time-wise difference between events coming in from the *from* and *to* routes are computed. The outgoing value is routed to a data block that represents a list of values. A weekly trigger (round shape) causes a computation of the list's average and the export of this information to the global outlet *Export*. This example shows the combination of two system events to a complex data source. With the traditional logging approach the two system events would be spread all over the entire log file and would have to be filtered out using appropriate data mining techniques. Then the events would have to be paired, subtracted and averaged. Compared to that, the application of a-priori knowledge about a systems behavior using observation facilities can provide superior results with less effort.

5. Self-observing system architecture

The definition of observation with the visual editor (see figure 3) results in a specification file that can be loaded and executed by all observation components inside products. The file is published using the global observation server and will be given a unique identification number in order to track which model was later on responsible for which collected data. Figure 4 shows the distribution of the overall observation system. Gradually the local product units (in figure 4 below the global observation unit) are updated with the new instructions and will from then on start to monitor respective events. The depicted product instances represent observation components that are integrated into products and execute this model inside a specialized runtime environment. The degree of integration determines many aspects of the overall system, for instance, how many concurrent events can be monitored or how the communication with the global unit takes place.

The integration of observation components into products depends on various factors that have to play

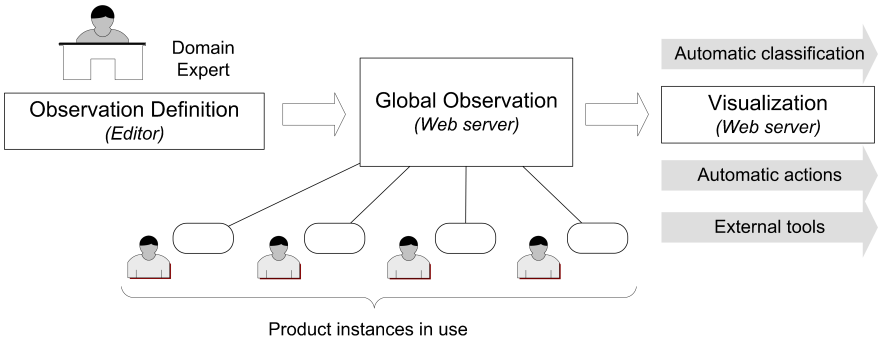

Figure 4. Overview of observation system

together in order to make this technology work. As there is a lot of variability among applicable systems, it is necessary to emphasize modularity that helps to bridge certain gaps in the implementations. The description of detailed integration mechanisms is out of the scope of this paper, but will be part of a later work, also introducing a design method for such systems. More information about the architecture can be found at [17].

The data that is collected and sent to the global unit is aggregated and stored in a database. From this point different scenarios are imaginable: the data can be visualized using charts that are updated in real-time, it can be analyzed using process mining techniques [18]. Also, events can be bound to alerts that are raised whenever a certain combination of low-level product events occur. Furthermore future uses include the incorporation of selected parts of the usage data into help-desks and repair shops in order to provide a better service quality to customers.

6. Case study

Our approach towards user modeling is currently being validated in collaboration with a large industrial partner. The consumer electronics products that are extended with self-observation facilities are in the alpha phase of development and serve as demonstrators that are given to key testers. These people use the "extended" products at home and will in the end also report subjective experiences. In the test, there are about 20 machines used in 8 countries world-wide with a collection of 15.000 to 28.000 data items daily depending on the actual usage behavior. Because of the privacy-intrusive nature of the test all users have been instructed about the conditions and accepted the fact that user-related data is collected.

The observation infrastructure that is in place consists of an observation authoring application which is based on the Eclipse platform [19]. A webserver provides both the configuration and data collection from local units using HTTP requests for communication. Additionally collected data is visualized using a web charting component (for structure see figure 4). The observation components integrated into the local products are implemented by using the *.NET* framework and hook into *Windows*-based systems. For the current industrial case we decided on the approach to execute the observation definition directly inside a runtime environment which emphasizes the intended reconfigurability. This runtime environment is designed in such a way, that it can be used also in future versions of the framework and also as an integral part of the design method.

When the case-study started the product hooks were only partly in place. Still, product developers were convinced easily to integrate more hooks into the product. In collaboration we built a hook interface for application-specific hooks and tested the system successfully using the visual authoring environment. Although the study is currently in progress, valuable findings throughout the development could already help to improve the integration of observation into pre-existing products. It became evident that developers showed the most interest in the visual authoring enviroment to define what should be observed. Whereas quality engineers informally agreed on the type of data to capture and then delegated to the developers in the team. While not being interested so much in the exact definition of retrievable information, the presentation of collected data was a major concern. This expresses that the current design of the visual language as an abstraction layer above an event-based system still is part of the technical domain. The key concepts match the thinking of observation stakeholders less than expected. This poses new requirements for future iterations of the visual language to build a bridge

between the technical domain of observable events and entirely data-related presentation methods.

Todays products are more often developed in small teams that merely specify the product and outsource well-defined subtasks. When the members of such a small product unit, managing the product development over the entire life cycle, develop an understanding for the importance of observability, they can work together closely on a specification of self-observation.

7. Conclusion

Regarding the fact that products are currently not designed for observation, we are working in the direction of a design method for self-observing systems. Therefore we developed an experimental framework for specification and implementation of observation functionality. A new visual specification language has been developed and simplifies the task of product usage data collection. We are in the phase where we try to get experience with this language and validate it. First usage results look very promising. We see that the availability of structured usage data that is captured directly inside of complex products gives options to collaborate and to apply knowledge. Especially the area of soft-faults and user modeling in general can benefit from this. For understandable and even adaptive products a strong observation foundation is needed to get insight in how the user currently uses a system and what his perceptions might be.

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] E. den Ouden, L. Yuan, P. Sonnemans, and A. Brombacher, "Quality and reliability problems from a consumer's perspective: an increasing problem overlooked by businesses?" in *Quality and Reliability Engineering International*, vol. 22, no. 7, 2006.

[2] A. Koca, A. Schouwenaar, and A. Brombacher, "Analysis of user-centered failure mechanisms in new product development for quality improvement," in *Proceedings of the 14th International Annual EurOMA Conference*, Ankara, Turkey, 2007.

[3] A. Kobsa, "Generic user modeling systems," *User Modeling and User-Adapted Interaction*, vol. 11, no. 1, pp. 49–63, Mar. 2001.

[4] M. Yudelson, T. Gavrilova, and P. Brusilovsky, "Towards user modeling meta-ontology," in *User Modeling*, 2005.

[5] D. Godoy and A. Amandi, "User profiling for web page filtering," *IEEE Internet Computing*, vol. 9, no. 4, 2005.

[6] N. Fine and W.-P. Brinkman, "Informing intelligent environments: creating profiled user interfaces," in *EUSAI '04: Proceedings of the 2nd European Union symposium on Ambient intelligence*, 2004.

[7] M. Diep, "Profiling deployed software: Assessing strategies and testing opportunities," *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, 2005.

[8] 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*, 2005.

[9] H. Lieberman, F. Paterno, and V. Wulf, *End-user development*, H. Lieberman, F. Paterno, and V. Wulf, Eds. Berlin : Springer, 2006.

[10] B. A. Myers, "Visual programming, programming by example, and program visualization: A taxonomy," in *Visual Programming Environments: Paradigms and Systems*, E. P. Glinert, Ed. IEEE Computer Society Press, 1990.

[11] P. N. Johnson-Laird, "Mental models," pp. 469–499, 1989.

[12] K. Ehrlich, *Mental Models in Cognitive Science: Essays in Honour of Phil Johnson-Laird*, 1996, ch. Applied Mental Models in Human-Computer Interaction.

[13] E. Karapanos and J.-B. Martens, "Characterizing the diversity in users' perceptions," in *Proceedings of Interact 2007*, Rio de Janeiro, Brazil, 2007.

[14] E. Rich, "Users are individuals: individualizing user models," *Int. J. Hum.-Comput. Stud.*, vol. 51, no. 2, pp. 323–338, 1999.

[15] D. C. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[16] G. Widmer and M. Kubat, "Learning in the presence of concept drift and hidden contexts," *Mach. Learn.*, vol. 23, no. 1, pp. 69–101, 1996.

[17] "The D'PUIS framework," 2007. [Online]. Available: www.softreliability.org/dpuis

[18] B. van Dongen, A. de Medeiros, H. Verbeek, A. Weijters, and W. van der Aalst, *Applications and Theory of Petri Nets 2005*, 2005, ch. The ProM Framework: A New Era in Process Mining Tool Support, pp. 444–454.

[19] "Eclipse," 2007. [Online]. Available: www.eclipse.org