Model-driven Instrumentation of graphical user interfaces.

Citation for published version (APA):
Funk, M., Hoyer, P., & Link, S. (2009). Model-driven Instrumentation of graphical user interfaces. In *Proceedings of the 2nd Internatinal Conference on Advances in Computer-Human Interactions, ACHI'09, 1-7 February 2009, Cancun, Mexico* (pp. 19-25). IEEE Computer Society. https://doi.org/10.1109/ACHI.2009.16

DOI:
10.1109/ACHI.2009.16

Document status and date:
Published: 01/01/2009

Document Version:
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 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

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.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 23. Sep. 2021

2009 Second International Conferences on Advances in Computer-Human Interactions

Model-driven Instrumentation of Graphical User Interfaces

Mathias Funk
Department of Electrical Engineering
Eindhoven University of Technology
5600MB Eindhoven, The Netherlands
m.funk@tue.nl

Philip Hoyer, Stefan Link
Research Group Cooperation & Management
Universität Karlsruhe (TH)
76128 Karlsruhe, Germany
{hoyer, link}@cm-tm.uka.de

Abstract

In today's continuously changing markets newly developed products often do not meet the demands and expectations of customers. Research on this problem identified a large gap between developer and user expectations. Approaches to bridge this gap are to provide the developers with better information on product usage and to create a fast feedback cycle that helps tackling usage problems. Therefore, the user interface of the product, the central point of human-computer interaction, has to be instrumented to collect accurate usage data which serves as basis for further improvement steps. This paper presents a novel engineering approach that combines model-driven user interface development and flexible instrumentation with run-time monitoring. In its application, it enables observation integration into products which provides comprehensive data about usage and thus allows for fast feedback cycles and consequently increased software quality. A case-study demonstrates the applicability of this approach.

1. Introduction

Current software intensive systems, ranging from complex consumer electronics to business applications, provide a broad functionality the user can access through a user interface. Research has shown that some user interfaces hinder an easy and fast usage of the product due to a user interface design that does not match the user's expectations. The problem can often be narrowed down to a gap between users' and designers' expectations [13]. In addition, nowadays software systems need to be flexible and adjustable to any number of different platforms or devices and are often used in fast changing business processes [3]. The satisfaction of a user working with a software system is a key indicator whether, e.g. a task within a business process is executed correctly and in appropriate speed. Still, information about usage data is seldomly collected by default. Therefore it is crucial to acquire accurate information in a fast way and to establish information feedback cycles between early testing and development. That way, user interfaces become possible which satisfy the user and enable a fast and proper execution of tasks.

The approach described in this paper which addresses the gap mentioned above is motivated in two ways: first, there is the need for *reliable and structured data about usage of user interfaces* which is rarely available. Second, to provide this data, *substantial effort to build in the necessary facilities* is required which is not feasible in most current product development processes. Focusing on graphical user interfaces (GUIs), low-level data is traditionally collected using runtime monitoring; often only basic user-system interactions like key presses, mouse movements, the use of external devices and the name of the active application or currently used system functions are retrievable. This data, although being objective and reliable, provides nothing more than a blurred picture of usage. It lacks context and make lengthy post-processing of the captured data necessary in order to retrieve meaningful information. Using traditional logging methods, much effort is required to build logging facilities as well as to tailor the logging to the needs of information stakeholders. Even then, the development of appropriate user interfaces for a certain group of users is an iterative process that involves specification, prototyping, testing, change of specifications, new prototypes, subsequent testing and so forth. Hence, there is a need for both flexible logging components and a development process which allows for changes on various levels of abstraction and quick builds of new prototypes.

Consequently, a model driven development approach for graphical user interfaces directly instrumented with

978-0-7695-3529-6/09 $25.00 © 2009 IEEE
DOI 10.1109/ACHI.2009.16

observation functionality is presented in this paper. The integration of observation aspects into the development process leverages the capturing of usage information in form of early models which provide access to the user interface and its inherent task hierarchies on a high level of abstraction. This enables the collection of semantically structured data throughout the user interface. At the same time, the approach aims at a high degree of automation. Both aspects, the GUI itself and its observation functionality are captured within models as central development artifacts. While capturing information at the right level of detail reduces the overall complexity of development, automated transformations in-between the different system models ensure a fast path towards implementation. This results in a flexible development process that enables quick iterations.

The remainder of this paper organized as follows. First, related work is presented and subsequently background information is given on the two techniques, model-driven GUI development and model-driven observation integration, that are linked together. The following section explains the combination of both techniques and also demonstrates the development flow by means of an example. This paper ends with a conclusion and an outline of future work.

2. Related Work

The need for appropriate models for graphical user interfaces (GUIs) has been observed in several approaches. An approach of Pinherio da Silva et al. [4] introduces new elements to UML with specialized symbols and new stereotypes. This extended UML is called UML for Interactive Systems (UMLi). UMLi allows for the modeling of certain GUI elements together with their behavior. Those elements are quite similar to the "GUI Profile" presented in [14]. However, the UMLi approach mainly focuses on the design phase of a software development process; regarding GUIs, it omits the artifacts which already could be acquired in earlier phases (e.g. the business modeling phase).

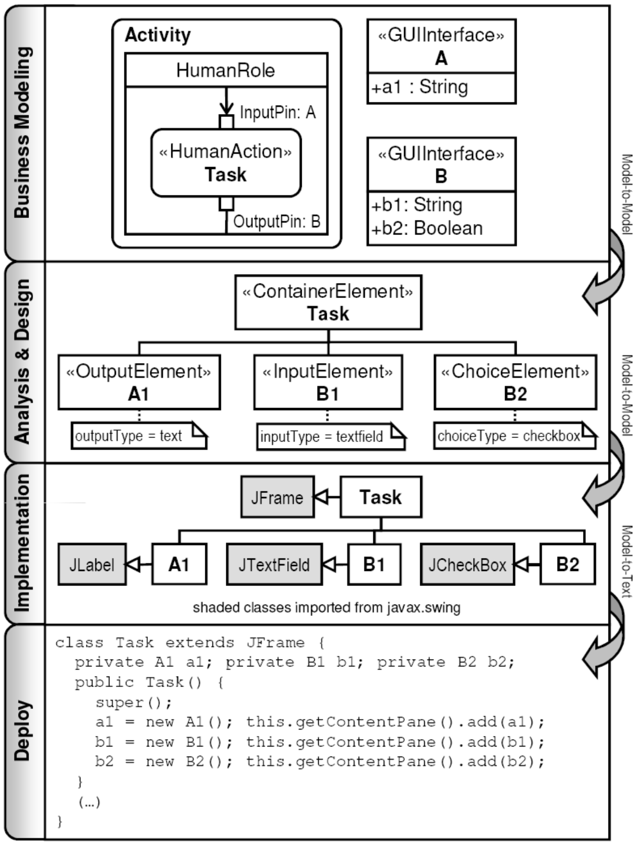
The approach of model-driven prototyping user interfaces is taken by Memmel, Bock et al. [15], incorporating three different models, very similar to those defined by the MDA [16]. The approach further provides a specific tool chain, dividing the UI development into layout, content and behavior, each specified by a dedicated tool and stored as a formal specification in XML. Compared to this approach making use of XML and specialized tools, we have taken a more general approach using UML Profiles [19]. Since any UML tool capable of UML Profiles can be used, no specific tools are required.

Sukaviriya, Sinha et al. [21] are likewise targeting the prototyping of GUIs. They present business processes as the first step in a business-driven software development process, using tool-independent concepts though modeled with a proprietary tool. While this tool allows taking four different views on the UML model which uses a UML Profile for user interface relevant data, we prefer using UML Activity Diagrams for modeling the business processes.

The domain of remote information retrieval is diverse. On the one hand there are attempts to study specific products or application domains such as building automation, mobile systems and the large area of web-monitoring [8, 9, 12, 20]. These monitoring systems are rather lightweight and use integrated logging facilities. The systems are by definition specialized and not applicable easily to other problem areas. This takes into account that they are build mostly for the exploration of usability problems and not designed for a large-scale use within potentially hundreds of logging units. On the other hand there are large frameworks like the WBEM [22, 10] which aim at monitoring enterprise business processes. The main focus of such systems is the business of a company as such, not products or product families. Typically, these systems involve a large implementation effort if used for observation tasks. They are huge systems that scale well, but specialization on product observation tasks involves too much effort to be practical and causes functionality overhead. As different as these areas might seem, the architecture of systems is surprisingly similar and basically leverages a client-server model together with integrated data collection facilities on the client side. Our approach to observation aims at preserving the strengths of both areas: fast and light-weight implementation, a high degree of automation, real-time feedback and flexibility. In addition we aim at an engineering process for observation systems helps practitioners build such systems for several applications or platforms.

3. Background

The complexity of software systems increases as new requirements like a continuous support of complex business processes arise. Modern business processes comprise tasks performed by humans which do not only require adequate IT-support but also need to be controlled and managed in their execution. Hence, existing process models in software engineering have to be improved to cope with such requirements as an integral part. The approach of model-driven software de-



```
class Task extends JFrame {
  private A1 a1; private B1 b1; private B2 b2;
  public Task() {
    super();
    a1 = new A1(); this.getContentPane().add(a1);
    b1 = new B1(); this.getContentPane().add(b1);
    b2 = new B2(); this.getContentPane().add(b2);
  }
  (...)
}
```

Figure 1. Process model extension for graphical user interfaces

velopment (MDSD) aims at achieving these improvements by highlighting the modeling of a software system as the core of any software development process [2]. Model-Driven Architecture (MDA) [16] is a well-known instance of MDSD; it permits to specify a software system through models on a very abstract level. By means of model-to-model transformations, these abstract models are subsequently transformed stepwise to more specialized models with a lower level of abstraction. Finally, source code for the desired platform is generated.

3.1. Process Model Extension for Graphical User Interfaces

The model-driven development of platform independent GUIs using a modern, iterative process model for business process management requires new meta-models or extensions to existing meta-models already in use. In our approach we utilize the UML [18] and the concept of "Profiles" [17] to create new modeling elements by extending existing UML meta-classes. Since most development processes use certain phases to address different GUI artifacts, our approach uses two UML profiles. Each profile is applied to a UML model being in particular use for a certain phase of the development process (cf. Figure 1).

The first UML profile named "GUI Activity Profile" is used during the platform independent business modeling, which is usually the beginning of a business-driven development process [3]. In this phase, the business process is modeled as a UML activity diagram with the participating roles and the associated business objects. During this phase, three stereotypes are used. First, the stereotype "HumanAction" extending the UML meta-class "InvocationAction" is used to allow the business analyst to mark those steps in the business process requiring human interaction. Second, since the final GUI model as described in the well-known MVC pattern [1] is already known in this phase, we model it using one or more Interfaces with the stereotype "GUIInterface". GUIInterfaces only contain attributes relevant for the GUI. Input and/or output pins owned by the HumanAction link to the GUIInterface via its type. Third, the outgoing flows of a HumanAction are stereotyped as "ActionFlow". An ActionFlow acts as a navigational element, which closes or leaves the GUI, submits the entered values and returns control to the workflow (cf. Figure 4).

The next step is done during the analysis and design phase by a GUI expert who specifies the view of the GUI using the "GUI Profile". In order to representing the basic elements of a platform independent GUI view, the profile introduces the stereotypes "InputElement", "OutputElement", "ChoiceElement", "ContainerElement" and "ActionElement" extending the UML metaclass "Class" and providing tagged values for those (refer to [14] for a detailed description). The elements are generated by a model-to-model transformation using the business process with the GUI Activity Profile described above as the source model.

The transformation seeks for "HumanAction" tasks (cf. Figure 1) and transforms the attributes of the linked GUIInterface to stereotyped classes; the used pin and attribute type determine the stereotype of the generated class. For example, GUIInterface attributes used only for input pins result in read-only data which is display to the user at runtime. Therefore these attributes are transformed to GUI output elements. In contrast, attributes used with an output pin or both types of pins can be edited in the final GUI, thus a transformation to input elements. Besides that, ActionFlows are transformed to ActionElements which usually result in buttons in the final GUI.

The platform independent GUI model is transformed to a platform specific GUI model by a second model-to-model transformation, using a platform model like Java Swing [11]. A Java Swing expert can further refine the generated platform specific GUI model with platform specific details. Finally, the platform specific GUI model is transformed to deployable GUI code.

3.2. Model-driven Observation Integration

The observation approach as described in [6] is a model-driven technique for data collection. This technique addresses the challenge of *collecting (usage) information from remote product instances.* These instances are distributed to more natural usage environments than company-based testing facilities. Based on this setting, the observation approach allows information stakeholders such as knowledge engineers, quality engineers, interaction designers, and product managers to *update the way in which the data is collected remotely and on-the-fly.* This enables a novel iterative usage data collection process, very much in line with current product development practices.

Observation model and semantics

A formal model of observation, that is, a detailed specification what should be observed and how collected data should be processed, is the main artifact of the technique. This model communicates observation logic defined by information stakeholders via a layer of observation management proxies towards the product instances which carry out the actual data collection. Hence, observation components inside product instances act according to the specification of observation given in the model.

The model-driven observation integration leverages that the GUI is specified continuously in the form of models, from high-level GUI tasks to implementation level. This high-level description captures more semantics about GUI activities than very specific models or even plain code. For instance, a single mouse click event that could be observed by low-level observation integration has no meaning without a proper context. The same event observed using a elaborate GUI integration can not only provide the window element that was triggered, but also the application context and status in which the interaction option was enabled, leading to insight about the current high-level GUI task that was performed.

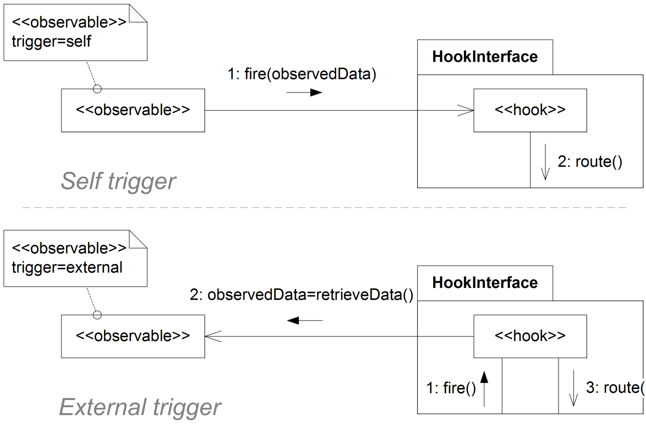


Figure 2. Observable-Hook trigger patterns

Observation profile

While the runtime configuration and distribution of observation specification tasks are carried out using the model interpretation technique [5], the integration of observation into the product is done in a more traditional model-driven way [7]. A long as the product development and the instrumentation for observation are performed in two separate development flows, both the applicability of the approach and the depth of observation integration is *clearly limited.* If product development and observation integration are combined, the benefits are not only a better reusability of existing parts in future observation scenarios, but also the reduced effort within one modeling domain. In this case, the observation integration can use the captured semantics in early models to weave dedicated observation facilities in. That said, the full benefits of using observation in a system are only possible if also the development of such a system is done using model-driven techniques and vice versa.

The core of the model-driven observation integration process is a UML profile. This *observation profile* essentially provides a vast vocabulary for the development of an observation system. It is divided into five main sub-profiles that constitute the three different layers of an observation system: *authoring and analysis* layer, *management and repository* layer, and *execution* layer. In this context, only the *integration* part of the *execution* layer is of interest and will be briefly described in the following. The actual data collection within a product instance is carried out using a component added to the host system. This component has to interface the host at various places in order to acquire the sought-after data. Therefore, the concept of *hooks* has been defined to express a (virtual) place in a system where data can be perceived. From the observation point of view, hooks serve as proxies that encapsulate certain (raw) data sources in the host system.

The second important concept is the *observable* item which annotates an element of the host system that provides observable data and that shall be connected to a respective hook element. The combination of observable item and hook defines the interface between host system and observation system and constitutes a specialized form of communication that can be expressed in form of patterns.

Since, hooks can be triggered to record data or can simply delegate system events, the two interaction patterns are shown in Figure 2. The first pattern, *self-triggering*, simply expresses that the hook reacts on an event coming from the host system and delegates it to the observation system for processing. In contrast, the second pattern, *externally triggered*, expresses that the connected part of the host system can be triggered, e.g. periodically, by the observation module in order to sample events from a continuous data stream. This pattern enables data retrieval from sources in the host system which are costly to assess or which provide meaningful information only in certain circumstances.

4. Model-Driven Instrumentation

The model-driven instrumentation flow extends the model-driven flow as described in Section 3.1. Figure 3 shows an overview on the development flow as used in this approach. Integrating the observation into a system means to link places in the host system that offer data via hooks to an observation component which handles data processing and transmission. This can be achieved seamlessly using the models of the GUI in different levels of abstraction. The model-driven instrumentation process is divided into three main steps: After the initial capturing of GUI semantics in platform-independent and platform-specific models, this information can be used to (1) *identify the set of observable items* within the application. In a subsequent step, (2) matching *hooks are created for all observable items* and linked. In the following model-to-text transformation step, the set of observable-hook pairs are used to (3) *generate interaction code defined by the respective trigger pattern* (cf. Figure 2).

Observation is introduced into the modeling of the GUI during the analysis and design phase. Although it is possible to add observation later in the process, it is advisable to do this as early as possible, since this largely affects the amount of observation in the application, but also the level of semantic structure and meaning the collected data might contain. The earlier observable elements are annotated in the model, the more natural context is given to data.

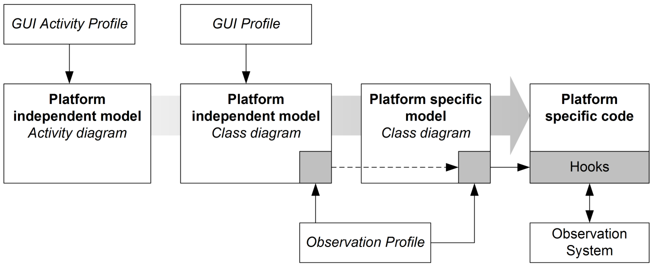


Figure 3. Model-driven instrumentation process overview

This extension of the original process for GUI development introduces only a small number of new elements during the modeling phases. In addition, several observation support components that handle communication, configuration and data processing can be modeled with the help of the observation UML profile. The profile proposes an observation system that is structured such that reuse of this supportive system is encouraged. This leads towards an observation platform architecture which remains stable is largely independent from the actual number and properties of observable items in the host systems.

As soon as this is accomplished, the interference of the observation system with the original GUI modeling process or the additional observation modeling means is minimal and non-invasive. It suffices to incorporate and use the *observable* concept during the specification and design time - with the appropriate transformation support - all other tasks necessary for observation integration are taken care of.

The benefits of this approach can be summarized as follows: The instrumentation of GUI elements provides straight-forward access to semantics that are tedious to achieve or even unavailable using traditional logging approaches. The annotation of observable elements already on the abstraction of high-level tasks automatically traces semantics to implementation. Furthermore, the model-driven GUI instrumentation process separates the concerns of observation annotation and platform development. While the former task can be carried out repeatedly, often necessary with changing observation needs, for the latter task it suffices to build a platform implementation once. The advantages of this separation become evident even within one product. In case of product families, observation system reuse enables to quickly introduce observation into several different systems in parallel. Finally, the process is easy to automate; changes in an abstract model can be tracked via automatic transformation to the implementation.

In the following section, the model-driven instrumentation process shall be described in more detail by means of an example.

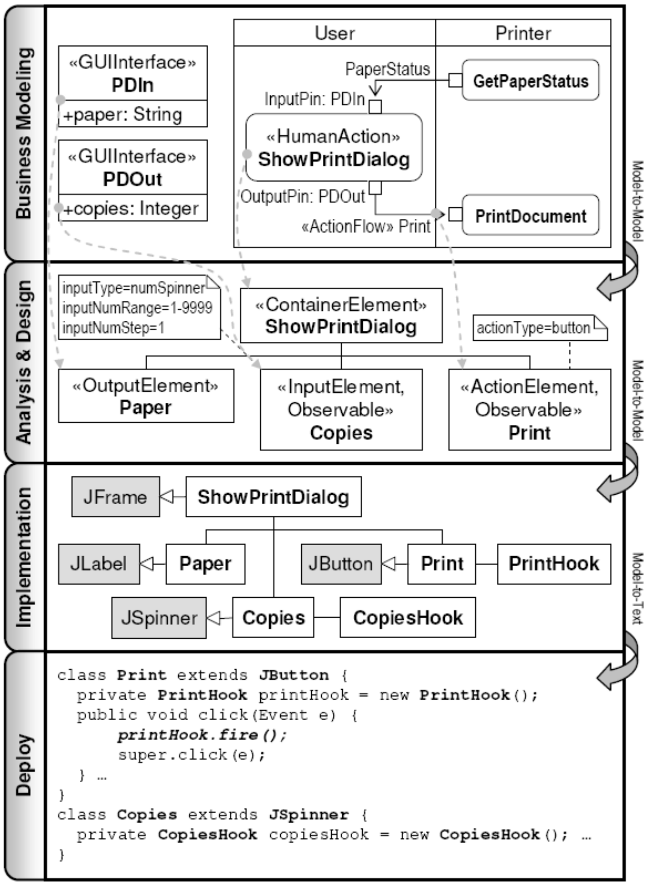
5. Case Study

In this section, the concept is exemplified with the prototype of a group collaboration application that was designed to help an organization plan smaller projects. The application consists of a clients running on the users' personal workstations, and a server component that stores all data for backup and synchronization purposes. Among other features, the client application permits the users to print out personal worksheets and weekly timetables. Several prototypes of this application shall be tested for several months within an office environment, and observation has to be introduced into the system. A simplified version of the print dialog serves as an example to apply the model-driven GUI instrumentation flow. Figure 4 shows the different phases of development.

During the *business modeling* phase the user interface is modeled in terms of abstract tasks. Relevant input and output elements such as "paper status" and the "number of copies" are also shown.

The first actual instrumentation step takes place in the *analysis and design* phase and consists of simply labeling all items that should be observed, i.e., adding the stereotype «Observable» to the class. Within the ContainerElement "ShowPrintDialog" two of three child elements are annotated and thus marked for observation: "Copies" and "Print". The first element is an input field for the number of copies, so the observable data is simply the number of copies requested by a user. The second element is an ActionElement and the observation thereof triggers an event as soon as the print job is started.

Subsequently, observable items can be identified clearly and handled by specialized transformation rules which add respective «Hook» elements to the *implementation* model of the system. The more detailed implementation model provides the necessary building blocks for linking observable system parts with the observation system. According to the annotation of the "Copies" and the "Print" elements, new hook classes "CopiesHook" and "PrintHook" are created and linked to their respective observable counterpart.



```
class Print extends JButton {
  private PrintHook printHook = new PrintHook();
  public void click(Event e) {
      printHook.fire();
      super.click(e);
  } …
}
class Copies extends JSpinner {
  private CopiesHook copiesHook = new CopiesHook(); …
}
```

Figure 4. Model-driven instrumentation process

Finally a model-to-text transformation takes observable - hook pairs as input and generates special interaction code into the observable elements classes. Basically, function calls from or to hooks are generated together with the rest of the system. The example shows that the "Print" class contains a "PrintHook" object and calls the $fire()$ method whenever a click event is detected.

The example shows that the process model-driven GUI instrumentation introduces observation at an abstraction level where it is easily manageable and where interference with other modeling tasks is kept at a minimum. This enables rapid iterations for several different versions of the same system. Different GUI concepts and also interaction patterns can be tested easily.

6. Conclusion and Future Work

The specification, design and finally testing of user interfaces is a challenging part of development that is often overlooked, neglected, or spent insufficient time on. The effort needed for testing a UI is crucial in current product creation processes, due to time-to-market pressure and complexity of nowadays products. Model-driven GUI development helps automate the flow from specification to prototype implementation and model-driven instrumentation of such interfaces finally provides access to data beneficial for UI evaluation. In this paper, observation integration by means of a model-driven technique is shown as a feasible approach towards both structured and reliable usage data. This novel technique enables to *quickly specify, generate and measure GUI* both for test use and within released products. Currently, modeling professionals have to do the labeling of observables during the analysis and design phase, whereas in the future, other options might arise: for instance, it is imaginable that actual information stakeholders are involved in the selection and annotation of observable system parts. Potentially, heuristic algorithms can be expected to identify items worth observing during system design and automatically generate set of potential hooks from the interface description.

Acknowledgments

Some of the authors are supported by 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. Burbeck. Applications programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC), 1987.

[2] G. Cernosek and E. Naiburg. The value of modeling. Technical report, IBM developerworks, 2004.

[3] P. Chowdhary, K. Bhaskaran, N. Caswell, H. Chang, T. Chao, and S.-K. Chen. Model driven development for business performance management. *IBM Systems Journal*, 45(3), 2006.

[4] P. P. da Silva and N. W. Paton. Improving uml support for user interface design: A metric assessment of umli. In *ICSE Workshop on SE-HCI*, pages 76–83, 2003.

[5] M. Funk, P. H. A. van der Putten, and H. Corporaal. Model interpretation for executable observation specifications. In *Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering*. Knowledge Systems Institute, 2008.

[6] M. Funk, P. H. A. 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*, Saint Luce, Martinique, 2008.

[7] M. Funk, P. H. A. van der Putten, and H. Corporaal. UML profile for modeling product observation. In *Proceedings of the Forum on Specification and Design Languages (FDL'08)*, page to be published. IEEE Computer Society, 2008.

[8] H. Hartson and J. Castillo. Remote evaluation for post-deployment usability improvement. *Proceedings of the working conference on Advanced visual interfaces*, pages 22–29, 1998.

[9] D. M. Hilbert and D. F. Redmiles. An approach to large-scale collection of application usage data over the internet. *icse*, 00:136, 1998.

[10] C. Hobbs. *A Practical Approach to WBEM/CIM Management.* Auerbach Publications, 2004.

[11] M. Hoy, D. Wood, M. Loy, J. Elliot, and R. Eckstein. *Java Swing.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[12] 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.

[13] E. Karapanos and J.-B. Martens. On the discrepancies between designers' and users' perceptions as antecedents of failures in motivating use. In K. Blashki, editor, *International Conference Interfaces and Human Computer Interaction*, IADIS, pages 206–210, Lisbon, 2007.

[14] S. Link, T. Schuster, P. Hoyer, and S. Abeck. Focusing graphical user interfaces in model-driven software development. In *Proceedings of the First International Conference on Advances in Computer-Human Interaction*, pages 3–8, Saint Luce, Martinique, 2008.

[15] T. Memmel, C. Bock, and H. Reiterer. Model-driven prototyping for corporate software specification. In *Proceedings of the EHCI-HCSE-DSVIS'07.* available online: `http://www.se-hci.org/ehci-hcse-dsvis07/accepted-papers.html`, Mar 2007.

[16] J. Miller and J. Mukerji. MDA guide version 1.0.1. Technical report, Object Management Group (OMG), 2003.

[17] OMG. White paper on the profile mechanism. Technical report, Object Management Group, April 1999.

[18] OMG. Unified modeling language. Technical report, Object Management Group, 2006.

[19] OMG. Unified modeling language: Superstructure, version 2.1.1. Technical Report formal/2007-02-03, Object Management Group, 2007.

[20] E. Shifroni and B. Shanon. Interactive user modeling: An integrative explicit-implicit approach. *User Modeling and User-Adapted Interaction*, 2(4):331–365, Dec. 1992.

[21] N. Sukaviriya, V. Sinha, T. Ramachandra, and S. Mani. Model-driven approach for managing human interface design life cycle. In *MoDELS*, pages 226–240, 2007.

[22] Web-based Enterprise Management (WBEM). online: `http://www.dmtf.org/standards/wbem/`.