UDC 004.3

M. BOŠKOVIĆ, T. WARNS, W. HASSELBRING

University of Oldenburg, Germany

MODEL DRIVEN INSTRUMENTATION FOR RELATIONAL EVENT TRACES*

Instrumentation of software is a part of debugging, performance evaluation and autonomic software systems. It enables the observability of program behaviour. However, instrumentation is costly and error prone. This paper presents an approach called *Model Driven Instrumentation for Relational Event Traces*. The approach enables the specification of system models and models for instrumentation as separate concerns, and allows to automatically generate instrumented systems from the models.

Instrumentation, Model Driven Architecture

Introduction

Development of software is based on defining the structure and the behaviour of programs. However, this is not sufficient for completely understanding the execution of a program. For such a complete understanding, more data about the execution itself, like time ordering of events or data flow, must be somehow observable. Observability of a program execution can only be accomplished by instrumentation.

Instrumentation of software means to insert probes into the system [1]. Usually, programs are instrumented with software probes. Software probes are pieces of code inserted for instrumentation of a program. They can be added automatically by using a variety of techniques (e.g. compiler modification) or tools (e.g. profiling tools), or they can be inserted during software implementation. Although automatic instrumentation of a program significantly reduces effort of developers or performance analyst, manual insertion has significant advantages.

Present research addresses instrumentation of software on different levels like source code modification or executable code modification. In our approach, we raise the level of abstraction for instrumentation. It enables model driven development of software functionality with integrated instrumentation.

*This work is supported by the German Research Foundation (DFG), grant GRK 1076/1

The paper is structured as follows. The next section presents a detailed description of instrumentation. Section 2 describes the basics of Model Driven Architecture. The core of the paper is Section 3 where the research idea is presented. In Section 4, related work is described. The last section contains conclusions and ideas for future work.

1. Instrumentation

Instrumentation is already used for monitoring system behavior in other disciplines like electrical engineering. A voltmeter and an oscilloscope are widely used for instrumentation in design and analysis of electrical circuits.

In software engineering, instrumentation is used for debugging, profiling/measurement and runtime surveillance/monitoring [2]. Debugging is the process of finding and removing faults in a program. Profiling is the process of collecting data about the overall execution of an application program or a part of it [3]. Measurement gives absolute numbers of the time spent for the execution of functions or about some resources used. Instrumentation is also used to enable runtime surveillance and monitoring. Monitoring is the process of collecting data about the execution and making it observable, while surveillance is collecting data and comparing it to some predefined values.

Because of the nature of a software artifact, there is

a variety of ways to instrument software. Software can be instrumented by using dedicated tools. They are distinguished into hardware, software, firmware and hybrid instruments [4]. Software instruments are programs that are used for instrumentation. Hardware instruments are external devices that are attached to the computer system with high resistance wires called hardware probes [1]. Likewise firmware can also be instrumented. Moreover, there are instruments which combine hardware, software and firmware instrumentation. These kinds of instruments are called hybrid instruments.

Besides tools for software instrumentation, there are alternative ways for automatically instrumenting a program, like a compiler modification or usage of pre-instrumented emulators.

Despite the fact that automatic instrumentation liberates a developer from manually inserting software probes, manual instrumentation has major advantages:

– precision – to gather exactly the data that is needed. It means to define which data to collect;

– granularity – to instrument only events which we are interested in. It means to define where to place probes;

– control – possibility of turning on and off desired probes;

– installation of tools – Installation of tools can be error prone;

– usage of tools – Developers need to be educated for the usage of tools

On the other hand, manual insertion of software probes increases source code complexity. Source code complexity makes a program harder to understand and maintain. Furthermore, manual instrumentation can often be error prone.

Generally, there are two techniques for data collection: program counter (PC) sampling and event tracing [3]. The program counter sampling is a statistical technique in which states of particular parts of a system, like a memory or a program stack, are sampled at particular points in time. An alternative to the PC sampling is the event tracing. The result of event tracing is a program trace. The program trace is a dynamic list of events generated by a program during execution.

The instrumentation can significantly increase the complexity of code and can be error prone. To reduce these shortcomings, we raise the level of abstraction to model level. Furthermore, we separate concerns of development of functionality and of instrumentation. However, to enable model driven development of a program and the instrumentation, the conceptual framework Model Driven Architecture [5] is used.

2. Model Driven Architecture

Models are used to express specifications of systems that should be made or that already exist. It is a set of statements about some system under study [6]. At the moment, the development of software systems is code centric. In this kind of development approach, models are only used to present some design ideas or for documentation.

However, models can be used for completely specifying software. Consequently, programs can be generated from such models. This kind of approach is called Model Driven Development (MDD) [7].

To support MDD, the Open Management Group–OMG (http://www.omg.org), a consortium of software vendors and users from industry, academia and government, offers the conceptual framework called Model Driven Architecture (MDA). MDA defines three viewpoints of a software model: Computation Independent Model (CIM), Platform Independent Model (PIM), and Platform Specific Model (PSM).

A CIM focuses on software requirements and does not care about the system structure. It should be made by domain experts.

A PIM is a computation dependent model but does not take into account a particular implementation technology.

A complete specification of a system is given in the PSM. The PSM is written in terms of some particular technology like .NET, Java or CORBA. This kind of model comprises a PIM with characteristics of the implementation platform.

OMG's MDA is based on a four layer metamodeling architecture and several adopted standards. The standards which MDA relies on are the Unified Modelling Language (UML) [8], the Meta Object Facility (MOF) [9] and XML Metadata Interchange (XMI) [9]. The UML profile [10] is also a standard which MDA relies on. It is a standard UML extension mechanism. The standards and the conceptual framework are presented in Fig. 1.

On the top of the architecture is the MOF, an abstract language for specifying metamodels. Metamodels are used to define modeling languages, like for instance UML. Furthermore, MOF metamodels can be used to extend existing metamodels and modelling languages.

As an example, we will model a real world entity Person with a UML class. The Person as a real world entity is placed on the M0 level. The Person can be modeled with a UML class. This model is placed on the M1 layer. The concept of a UML class is defined on the M2 layer. It is defined with the MOF which is placed on M3 layer.

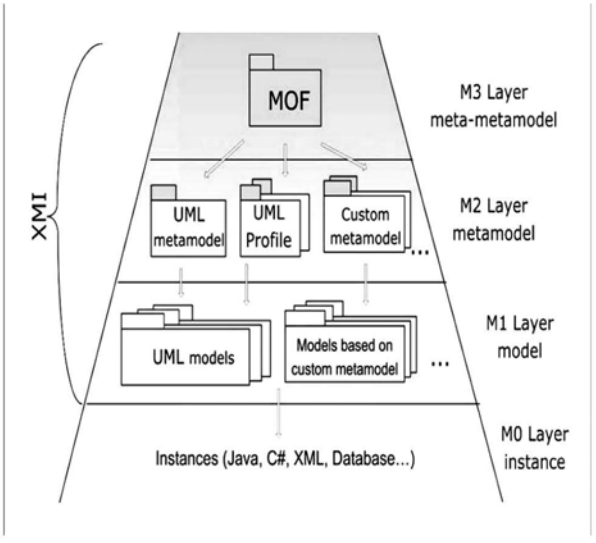

Fig. 1. The OMG MOF based conceptual architecture and standards on which it relies [11]

Model Driven Development enables developers to focus on functional requirements without the implementation details. However, for large systems where many system wide crosscutting concerns, like instrumentation, have to be addressed, this is not sufficient. For this reason, we want to separate concerns of the system functionality modeling and instrumentation.

3. Model Driven Instrumentation

The idea of Model Driven Instrumentation considers instrumentation as a separate concern on model level. The used technique for data collection is event tracing.

Event traces are structured in a relational manner like presented in [12]. Relations can be seen as tables of relational databases or tuples. Each relation consists of several fields. The data about the execution is stored in the fields. The relational approach enables analysis with declarative queries made in SQL-like languages.

The implementation of the approach is sketched in Fig. 2. A system modelling language allows specifying the software structure and the behaviour models, like class diagrams or state charts, and a event trace model for defining event traces.

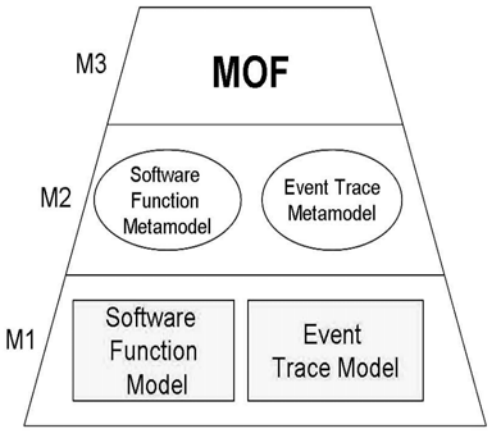

Fig. 2. Structure of implementation of the approach

Since we can have different modeling languages, we will develop a basic structure of event traces. The basic structure of event traces is relational and consists of an

identifier of a trace, name of a field, data type of a field and a operation that is related to a field. Operation is used for collection of some particular data during execution, and the data is placed in the field of relational trace.

The basic structure of traces is used for the definition of particular event trace metamodels on the M2 layer for instrumentation of models of different modeling languages. For different modeling languages (M2), different types of event traces (M2) and operations can be defined. The type of an event trace is defined for instrumentation of some particular element types of the target modeling language. This means that event trace model (M1) made according to some event trace type definition (M2) can be linked to an element in the software function model (M1) of the element type (M2) for which event trace type (M2) is made. Furthermore, for each event trace type (M2), a set of operations will be defined that can be used in a trace model (M1). Set of operations is defined according to the element type for which the trace type is defined. To better explain idea, we will use state charts.

In case of state charts, we could develop two types of event traces, event trace type for state, and event trace type for transition. For state event trace type, we could define operations for collecting the name of the state, and start and end timestamps of a state. In case of transition event trace operations for collecting the name of the event that fires the transition and the time of transition.

Definition of event trace model on the M1 layer according to some particular event trace type on the M2 layer consists of a definition of the names of fields, data types of that fields and operation related to that field. For example we can define the event trace model (M1) for states, named Trace1. The event trace model consists of fields named StateName, and Time. For these two fields we relate operations for getting the name of a state and the time stamp of the beginning of the state, respectively. Now this event trace model (M1) we can link to some particular states in the software function model (M1). This means that when a state that is linked to event trace model occurs during execution, it automatically produces one record with name and start time of the state in event trace.

The instrumentation and system generation is presented in Fig. 3.

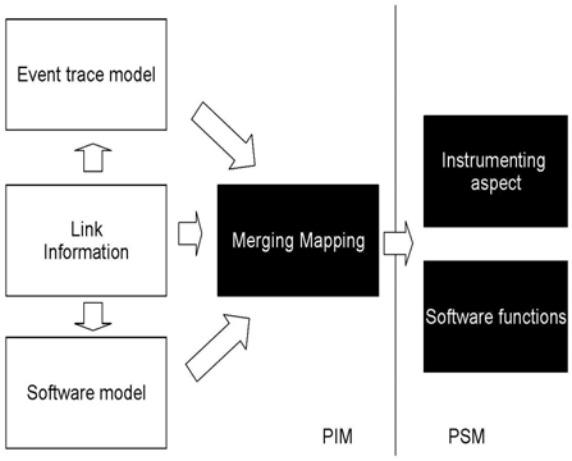

Fig. 3. Instrumentation and system generation

First the basic functionality of the system is made. For instance class diagram or state chart. After it is developed, a system performance analyst, or developer defines event traces. In this part software developer or performance analyst defines fields of event traces, names of the fields and operations that will be executed for data collection. When system and event traces are defined, linking between these two models is done. At this point we define where the data collection takes place. This means that we will link traces we defined for the elements of the model. Later, transformation from a model to code is done. For implementation platform we plan to use platform which enables Aspect Oriented Programming (AOP) [13]. AOP is methodology which enables separation of crosscutting concerns at code level. More details about AOP can be found in [13].

With the approach presented in this paper, we present foundation for instrumentation on model level. Furthermore, approach enables separation of concerns. Separation of concerns facilitates transparent

instrumentation of software and independent planning of event traces. Since event traces are in the relational form, it enables analysis of the execution in a declarative manner.

5. Related Work

Instrumentation is an important part of the analysis of program behaviour. It enables the extraction of data about the program execution. However, it can be costly and error prone because of increasing complexity of code. The present research addresses cost reduction of adding the instrumentation.

To enable instrumentation of software systems as a separate concern Aspect Oriented Programming is used in [2] and [14]. CORBA based applications can be instrumented as a separate concern with the usage of interceptors [15]. Another approach is to place a transparent software layer for data collection presented between execution platform and application. This kind of approach is presented in [16].

To enable instrumentation as a separate concern on code level several instrumentation languages are introduced like the Metric Description Language (MDL) [17] and the Program Monitoring and Measuring System (PMMS) [18].

The most related research is presented at [19]. In their approach first source code of application is developed. For instrumentation they develop monitoring model. Monitoring model consists of activities presented as nodes and time interdependence within activities presented as arcs. Each activity is unequivocally related with some module or procedure by its name. After the monitoring model is developed source code is automatically instrumented for time stamping. The purpose is to get the data about the time orderings of activities in the program.

In our approach data that will be collected can be defined. In addition, software models are used to define instrumentation points. Furthermore, instrumented system is automatically generated from models. Therefore instrumentation can be integrated into the process of software development and can be used for different analysis, not only time ordering.

Conclusion, Current Status and Future Work

Instrumentation of programs is very important for analyzing program execution. However because of intertwining the basic functionality with the instrumentation code, it can cause significant problems.

In this paper we presented a novel approach on instrumentation called – Model Driven Instrumentation for Relational Event Traces. The idea is to raise the level of instrumentation to the model level and to separate concerns of the instrumentation and basic software functionality. For the technique of instrumentation we have chosen the event tracing. Structure of event traces is relational because of the possibility of making declarative questions about program execution.

The research is in the stage of a PhD proposal. First of the next steps is the identification of data types and development of basic structure of relational traces. When it is developed we will make event trace metamodels for instrumentation of class diagrams and statecharts.

To enable automatic generation of code from models, transformations for these two modeling languages will be made. As a proof of concept we plan to implement a prototype tool for instrumentation on model level.

References

1. Ferrari D. Computer Systems Performance Evaluation, Prentice-Hall, New Jersey, USA, 1978.

2. Marenholz D. et al. Program Instrumentation for Debugging and Monitoring with AspectC++ // In ISORC'02: Proc. of 5th IEEE Int. Symp. on Object Oriented Real-Time Distr. Computing, Washington, DC, USA, IEEE Comp. Soc. – 2002. – P. 249-256.

3. Lilja D. J. Measuring Computer Performance: A Practitioner's Guide. – Cambridge University Press, Cambridge, UK, 2000.

4. Smith C.U., Williams L.G. Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. – Addison-Wesley, USA, 2001.

5. Meta Object Facility (MOF) specification v2.0, OMG document ptc/04-10-15, October 2004. – [Электрон. ресурс]. – Режим доступа: http://www.omg.org/cgi-bin/apps/doc?ptc/04-10-14.pdf.

6. Seidewitz E. What Models Mean" // IEEE Soft., Vol. 20(5). – IEEE Comp. Soc., 2003. – P. 26-32.

7. Selic B. The Pragmatics of Model-Driven Development // IEEE Softw., Vol. 20(5), IEEE Comp. Soc. – Sept. 2003. – P. 19-25.

8. Unified Modeling Language (UML) 2.0 Specification, Infrastructure, OMG doc. pct/04-10-14, November, 2004. – [Электрон. ресурс]. – Режим доступа: http://www.omg.org/cgi-bin/apps/doc?ptc/04-10-14.pdf.

9. MOF 2.0/XMI Mapping Specification, v.2.1, OMG doc. formal/05-09-01, November, 2004. – [Электрон. ресурс]. – Режим доступа: http://www.omg.org/cgi-bin/apps/doc?ptc/04-10-14.pdf.

10. Miller J., Mukerji J. MDA Guide (Version 1.0), OMG document ptc/04-10-15, Jun 2003. – [Электрон. ресурс]. – Режим доступа: http://www.omg.org/docs/omg/03-06-01.pdf.

11. Djurić D., Gasević D., Devedžić V. Ontology Modeling and MDA // Journal of Object Technology, Vol. 4(1), Jan-Feb 2005. – P. 109-128. – [Электрон. ресурс]. – Режим доступа: http://www.jot.fm/issues/issue_2005_01/article3 .

12. Snodgrass R. A Relational Approach to Monitoring Complex Systems // ACM Trans. on Comp. Syst. – ACM Press. – May 1988. – Vol. 6 (2). –P. 157-196.

13. Laddad R. AspectJ in Action: Practical Aspect-Oriented Programming, Manning Publications and Co., Greenwich, CT, 2003.

14. Debusmann M., Geihs K. Efficient and Transparent Instrumentation of Application Components using an Aspect-Oriented Approach // In 14th IFIP/IEEE Workshop on Distr. Syst.: Operations and Management (DSOM '03), Lecture Notes in Computer Science, Vol. 1867, Heidelberg, Germany, Springer, October 2003. – P. 209-220.

15. Li J. Monitoring of Component-Based Systems. – Tech. Report HPL-2002-25, Imaging Syst. Laboratory, HP Laboratories Palo Alto, 2004.

16. Diaconescu A. et al. Automatic Performance Management in Component Based Systems // In Proc. of the 1st Int. Conf. on Autonomic Computing (ICAC '04), IEEE Comp. Soc., 2004. – P. 214-221.

17. Hollingsworth J.K. et al. MDL: A Language and Compiler for Dynamic Program Instrumentation // In *PACT* '97: Proc. of the 1997 Int. Conf. on Parallel Architectures and Compilation Techniques, Washington, DC, USA, IEEE Comp. Soc., 1997. – P. 201-213.

18. Liao Y., Cohen D. A Specificational Approach to High Level Program Monitoring and Measuring // IEEE Trans. on Soft. Engineering, Vol. 18(11), IEEE Comp. Soc. – 1997. – P. 969-978.

19. Klar R. et al. Tools for a Model-Driven Instrumentation for Monitoring // In Proc. of the 5th Int. Conf. On Modeling Techniques and Tools for Computer Performance Evaluation.

Поступила в редакцию 20.02.2006

Рецензент: д-р техн. наук, проф. В.С. Харченко, Национальный аэрокосмический университет им. Н.Е. Жуковского «ХАИ», Харьков.