Enhancing Root Cause Analysis with Runtime Models and Interactive Visualizations

Michael Szvetits[1] and Uwe Zdun[2]

[1] Software Engineering Group
Information Technology Institute, University of Applied Sciences Wiener Neustadt
michael.szvetits@fhwn.ac.at
[2] Software Architecture Group
Faculty of Computer Science, University of Vienna
uwe.zdun@univie.ac.at

Abstract. Recent research shows that runtime models can be used to build dynamic systems coping with changing requirements and execution environments. As software systems are getting bigger and more complex, locating malfunctioning software parts is no trivial task because of the vast amount of possible error sources and produced logging information. This information has to be traced back to faulty components, which often leads to editing of code scattered over different software artefacts. With such a fragmented view, challenges arise in finding out the root cause of the unwanted software behaviour. In this paper we propose the usage of runtime models in combination with model-driven techniques and interactive visualizations to ease tracing between log file entries and corresponding software artefacts. The contribution of this paper is a repository-based approach to augment root cause analysis with interactive tracing views while maximizing reusability of models created during the software development process.

Keywords: root cause analysis, models, runtime, visualization

1 Introduction

Recent research proposes models used at runtime to realize adaptable and manageable systems. Runtime models provide software with additional capabilities to reflect on its own structure and adapt itself according to changing requirements and execution environments. This is different to traditional software engineering where models are artefacts created during the software development phase, models have no connection to the resulting executable software, and the software is usually modified by re-deployment of changed software components. In addition to this extra deployment phase, such changes are often performed on artefacts residing on a low abstraction level. To overcome these limitations, the running system can use a model-based self-representation which is causally connected to it [1]. A causal connection enables feedback from runtime information to software models and thus reasoning on a higher abstraction level. The type of model

used for information feedback depends on the type of reasoning: for instance, architecture models are suited for system-wide adaptation, whereas state machine models are useful for validations of execution flows. The reuse of model artefacts at runtime not only supports adaptation, but also platform independence [2], monitoring/simulation [2], rule enforcement [3] and error handling [4].

The amount of runtime data produced by many applications tends to be huge and is recorded in different forms. Relevant information has to be extracted and put into context with corresponding artefacts produced in the development phase to reveal the root cause of unexpected software behaviour. The purpose of such an analysis is to find out the root effect within a chain of undesirable effects occurred during software execution [5]. Since implementation details are usually scattered over different development artefacts, further challenges arise to offer an effective navigation from logging messages to affected software parts. Techniques are needed to filter runtime information for particular criteria of interest.

In this paper we propose a novel approach to combine visualization techniques with models used at runtime (and usually created in the design phase) to their mutual benefit during root cause analysis: Model artefacts are utilized as additional information during root cause analysis while visualization techniques allow to focus on particular sections of logging output of the running application. We accomplish this by using model-driven techniques in combination with interactive visualizations to enable tracing between log file entries and corresponding software artefacts. A customized code generator produces annotated code which handles logging without the need of human intervention. Furthermore, model artefacts are stored in a repository to be accessible by a log file tracer, although our approach is not limited to such data access. That is, model data is also accessible to external applications to open the system for future extensions. At runtime, recorded logging data is fed back to interactive analysis views and associated model elements to support root cause analysis on a higher abstraction level, thus closing the loop between the development and runtime phases.

This paper is organized as follows: In the next section we give an overview of our approach to combine reusable models with interactive visualizations to gain aforementioned benefits. In Section 3 we describe the model-driven part of our work which prepares runtime access of model information. Furthermore, we introduce interactive visualizations to analyze runtime information and present a prototype demonstrating our idea by using a model repository to trace log file entries to corresponding software artefacts. In Section 4 we give an exemplary scenario for the usage of our approach. In Section 5 we discuss our results and lessons learned. In Section 6 we take a look at related work in terms of model feedback and interactive visualization. We conclude in Section 7.

2 Approach Overview

Our approach proposes semi-automated, interactive techniques to provide feedback from runtime information back to the relevant software artefacts. More precisely, the tool proposed in our approach listens to log file changes produced

by the running system, extracts the information and puts it into context with corresponding model artefacts and their elements. As a result, models created in the development phase are utilized at runtime and act as a representation of system parts affected by specific log messages. We use a model-driven approach to generate the logging and tracing environment for the software where the amount of produced runtime information can be adjusted by the developer at design time (see Section 3.1). Furthermore, we use interactive visualizations to reveal the root cause of unexpected software behaviour by making dependencies between software parts explicit and providing improved navigation capabilities to model artefacts used at runtime (see Section 3.2). A summary of the development steps is depicted in Figure 1.

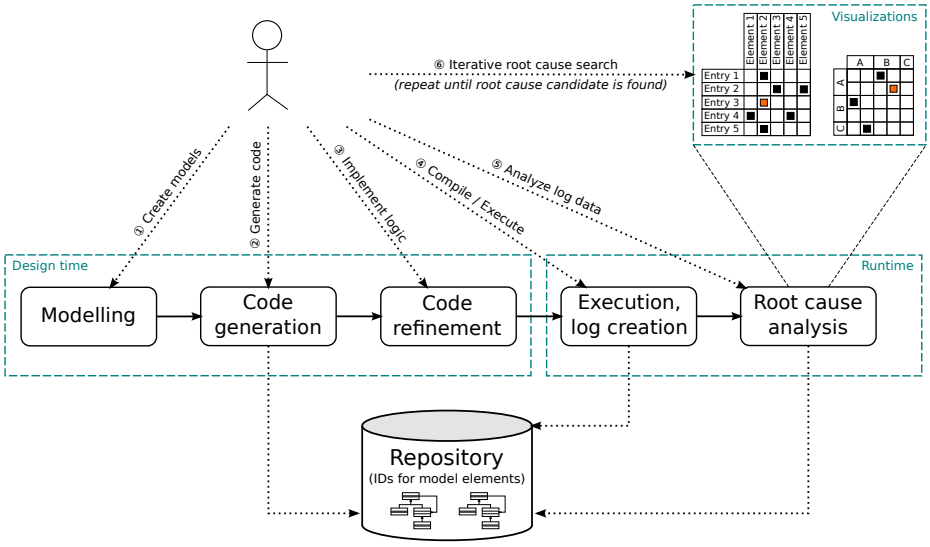


Fig. 1. User view of the development steps when using our approach

The first step of our approach is concerned with modelling of the system. In our approach, the modelling step can concern any model artefact that is compliant with a specific meta-model, although the current version of our prototype focuses on UML models only (see Section 4). These created model artefacts are utilized at runtime where runtime information is fed back to the models to find locations of errors and unexpected software behaviour. The next step deals with the development or adaptation of a code generator to produce model and logger code for the system. Furthermore, the code generator is responsible for storing models and their related elements in a repository which is accessible at runtime. This step requires no additional interaction of the developer and enables the resulting software to query its own structure and produce log file entries with a reference to the model that has created the software artefact producing the log

entry. This part of the development process leaves room for future extensions to augment the logger, e.g. to generate reports of software failures.

During code generation model-related code as well as logger code is created which emits log file entries in a format specified by the code generator. In the phase of code refinement, the actual software product is developed and the generated model code is integrated or extended by the custom implementation. Within custom code, log file entries can be emitted in the correct format by making calls to the previously generated logger code.

The combination of custom and generated code is then compiled to an executable application which acts as system under observation for subsequent analysis. Implemented logger calls obtain model information from the repository and produce log file entries traceable to the related model elements. The last step of the development is concerned with the analysis of the produced log files and the interactive visualization of the recorded data to enable efficient tracing between log file entries and corresponding model artefacts. In this phase, root cause analysis is an iterative action by narrowing down problems step-by-step with the help of multiple interactive visualizations. These visualizations highlight affected model elements and clarify their interrelationships.

As a summary, the key idea of realizing root cause analysis with interactive visualizations is a loop which is responsible for the feedback of runtime data to model artefacts created in design phase. We divide the steps into model-driven application development and feedback of runtime information with the help of interactive visualizations. Section 3.1 and 3.2 describe the details for these parts.

3 Approach Details

Based on the overview depicted in Figure 1, we give a more detailed view on the development process to provide insight into our model-driven development and interactive visualization parts.

3.1 Model-driven Runtime Preparation

We use model-driven techniques to prepare the resulting software product for runtime access of model-related logging data. Interactive visualizations are then used to process this data and provide root cause analysis capabilities. The first step is the creation of model artefacts to act as a base for code generation and information feedback. These model artefacts are then fed into a code generator which is responsible for two tasks: On the one hand, it has to store the models and its associated elements in a repository to be accessible at runtime. On the other hand, it must generate code out of the input models which is then refined by the developer. The generated code depends on the customized code generator and usually covers model-relevant code and additional components to enable logging. The generated logger code is responsible for writing log files in a format to be processable by a log file tracer or other external applications.

Within the repository, the model and all of its traceable model elements (e.g., packages, classes and methods) are stored and get a unique ID. In principle, our approach is independent of algorithms which generate such unique IDs – for our prototype, we used the hash code of the full qualified name of a model element (see Section 4). The assigned IDs are used by the code generator to create annotations for corresponding elements in the generated code. These annotations can be accessed by the logger component at runtime to produce traceable log files entries. Such entries usually consist of the related IDs and a generic message, but can be adapted by modifying the code generator itself.

In the next step, custom code is implemented which extends or calls the generated model code and realizes the actual business logic. Logging is controlled by annotating elements of interest or by calling the generated logger directly. Extending generated code is usually achieved by inheritance relationships between custom and generated classes in combination with overriding of generated operations. Depending on the code generator, a logger can evaluate such inheritance hierarchies at runtime through reflection to locate IDs of annotated elements and produce log entries containing references to associated operations and classes. This is an important step to close the loop between produced runtime information and the static structure modelled at design time since it enables interactive analysis views to provide navigation between log files and design artefacts.

For the dynamic aspects of the system, the logger component alone is not adequate since the logger class can only trace the caller of the logger itself, but not calls between different components (e.g., a statement in an annotated method calls another annotated method). We achieve tracing of such calls through aspect-oriented programming techniques: The code generator generates pointcuts (hooks to method invocations) for methods to be traced and advices (the actual hooking code) which call the logger with information about caller and callee.

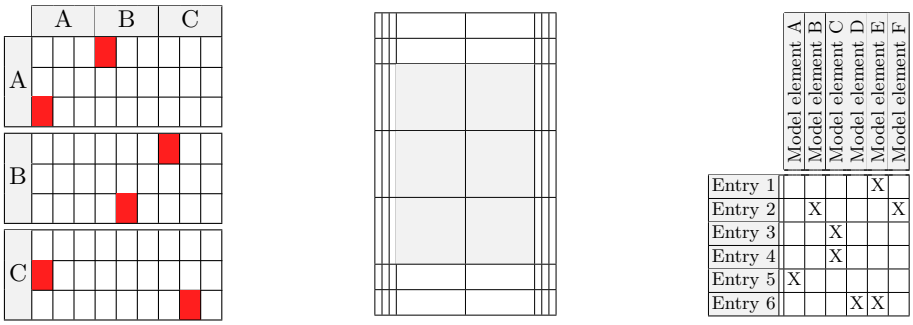
After compilation and execution, calls to the generated logging code will produce log file entries in a format which can be interpreted and fed back to model elements which have been created in the first step. For feedback of runtime information, a tracer component has to read produced log files and query associated model elements from the repository. To obtain live data from the application, the tracer must listen regularly to changes in these log files. In addition, it is also possible that the application itself queries the repository to obtain additional information about its own structure. The repository can also be queried from arbitrary clients which makes the architecture extensible for external tools.

3.2 Interactive Visualization of Runtime Information

After log entries have been read, interactive analysis views have to be populated so the user can select relevant data from the collection of runtime information. With visual support available, the amount of irrelevant messages can be minimized and affected components and their dependencies can quickly be found.

Van Ham [6] describes the use of *Call Matrices* as technique to visualize calls between components. A call matrix has the visualized components placed in its

row and column headers whereas matrix cells represent calls between the respective components (see Figure 2a). We use this kind of interactive visualization to display calls captured by our pointcut expressions to enable quick navigation to error locations and gain quick overview of connected components. More precisely, we put packages and their classes into row and column headers while matrix cells represent the methods of each class. A call can then be displayed by highlighting the associated matrix cell where the row stands for the calling and the column for the called method. In Figure 2a a call matrix consisting of three different components reveals that several errors occurred in calls between components (A-B, B-C and C-A) as well as within components (A-A, B-B and C-C). Different colors can be used to highlight priorities of call events, e.g. yellow for warnings and red for errors. Root cause analysis is thus supported in a way that the call of a faulty method can be traced back to the exact code location (file name and line number are provided by the cell data) while the caller can act as new starting point of further analysis. This backtracking can be done until no further logging data exists.



(a) Call Matrix [6] (b) Table Lens [7] (c) Log entry matrix

Fig. 2. Interactive visualization techniques, our approach utilizes (a) and (c)

Another form of interactive visualization is named *Table Lens* and is used by Rao and Card [7] to focus on particular ranges of tabular data. Table rows and columns can be adjusted by the user with the help of zooming and sliding to control the amount of displayed data. Rao and Card [7] define three operations to manipulate the currently focussed area: *Zoom* (change space allocated to the focus area), *Adjust* (change amount of content within the focus area) and *Slide* (change location of the focus area). Figure 2b shows a tabular data sheet where a range of 3 by 2 cells is currently focussed (cell content has been removed for the sake of simplicity). Our current prototype does not include this visualization technique, but is extensible to do so in the future (see Section 4).

Beside the call matrix, we implemented a matrix view where a simple list of all recorded log entries is set in relation to all corresponding model elements (see Figure 2c). Within this matrix view, sortable columns enable grouping of log

entries for particular model elements. Such grouping improves root cause analysis because hot spots with a high error density can easily be detected. Furthermore, if a faulty component has been detected with the call matrix approach, similar errors for the same component can quickly be revealed. Since the matrix grows rapidly with the number of log entries and model elements, this technique is only recommended for small projects. For a great number of data, one should fall back to Table Lens or a combination of Table Lens and log entry matrix.

To provide more insight into the structure and dependencies of a system, *Linking and Brushing* combines multiple visualizations by introducing synchronization mechanisms between them [8]. Changes performed on one visualization are communicated to all other visualizations to understand interrelationships between these artefacts, e.g. to find classes of an UML class diagram in other sequence diagrams. In our approach, when the user selects log entries of interest, encoded model-related information is extracted from the selection and displayed in corresponding model artefacts. This helps to quickly understand the impact of warnings/errors and to provide an efficient navigation between affected models.

4 Prototype and Exemplary Scenario

We implemented an Eclipse-based[1] prototype of our idea by using Papyrus as modelling toolkit, Acceleo as code generator and AspectJ for aspect-oriented capturing of dynamic calls. For log files, we included the file name, line number, a generic message and related IDs (packages, classes and methods) for all log file entries. The model repository is a relational database which stores the model file of UML models and holds references to each model element by their assigned unique identifier. In our prototype, we use the hash code of the full qualified name of the respective model element as unique identifier, although our approach is not limited to this technique as long as there is a unique identifier available. We implemented four different visualizations for more efficient root cause analysis which utilize the repository to trace selections back to the UML models:

- *List view*, a live view which lists all log entries in chronological order.
- *Matrix view*, a sortable view which relates log entries with model elements (see Figure 2c). Model element grouping enables hot spot identification.
- *Outline view*, a list of expandable log entries to view related model elements. Enables quick navigation from log entries to model elements.
- *Call Matrix view*, a view to trace dynamic calls between components (see Figure 2a). Reveals dependency chains of faulty components.

Figure 3 demonstrates our exemplary scenario which utilizes the matrix view and call matrix view to enhance root cause analysis. We consider a simplified class diagram of a linked list implementation to demonstrate the structure of our interactive visualizations. The scenario starts with the user recognizing unexpected software behaviour and starting the tracer component to constantly read the log file of the application and refresh the visualizations.

[1] http://projects.eclipse.org/list-of-projects

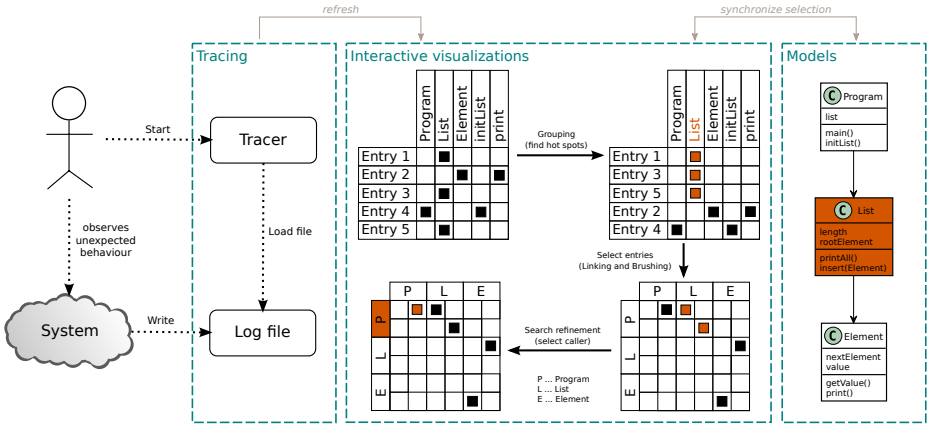


Fig. 3. Exemplary scenario of a step-by-step root cause analysis

Once the logging data has been read, the matrix view and call matrix view are populated with the data (other views are neglected in this scenario). Such views are not available in traditional logging approaches and contribute to a more efficient root cause analysis: Assuming an anomaly has occurred, the developer is able to take a first look into the matrix view to get an overview of all traceable model elements which were involved in log entries. Irrelevant model elements are not rendered as columns at all, providing a compact overview which is not available when analyzing log files by hand. For our example, the columns of the matrix view consist of three classes (*Program*, *List*, *Element*) and two methods (*initList*, *print*). The matrix view allows to group logging entries for suspicious model elements. If this grouping results in a large list of entries for an element, it is very likely that the error occurs within a frequently called method of that element. Element *List* depicted in Figure 3 is a candidate for such an element.

To inspect this element, the call matrix view is consulted to find out more information about called methods of this element (columns) as well as their respective callers (rows). This is directly achieved by selecting the entries in the matrix view since our implementation of Linking and Brushing [8] highlights all occurrences in other related views. The call matrix view enables direct navigation to model and code artefacts which contain calls of selected cells. Together with the generic log messages of entries, the user can identify possible error sources around these calls. If no suspicious spot can be found, the search is continued upwards the call hierarchy to the next caller which acts as new base for analysis. In case of our scenario, the next caller is the second method of class *Program* (that is, method *initList*, see left call matrix in Figure 3). For this simple example this could indicate that the list was filled with wrong values during initialization (e.g., null values) while unexpected behaviour was observed in the method *insert* in the first place. In general, this step is repeatable for subsequent callers in the call hierarchy until a root cause candidate is found (it is not repeatable for our exemplary scenario, since *Program* is the application entry point).

While navigating through log file entries with the help of interactive visualizations, our implementation of Linking and Brushing [8] is also responsible for highlighting model elements in existing UML models created at design time (see right side of Figure 3). On the other hand, we implemented an extension to the Papyrus model editor to realize Linking and Brushing the other way round, that is, the selection of elements in UML models highlights all associated log entries in the interactive visualizations. As a result, models created at design time are utilized at runtime, integrated into our approach and contribute to the overall understanding of the architecture while searching for anomalies. This synchronization mechanism between visualizations and models is used to monitor a running system if the tracer guarantees a live view on the logged data.

Regarding architecture, it is also possible that the tracer component receives the logging information remotely via network and interprets the information without further interventions of the user. Such connection between the system and the monitoring environment cannot be automated if logging messages follow an informal format, which is often the case in traditional logging. As a summary, the usage of interactive visualizations with runtime models exceeds manual root cause analysis in terms of logging effort, navigability, traceability and reusability.

5 Discussion

Our approach aims to combine the advantages of model-driven development with reusable software artefacts. Once the code generator is implemented, the system is extensible through the model repository and is fully transparent for the developer since manual actions are limited to logger calls where traceability is desired. All supporting artefacts (e.g. logger, database entries, annotations, aspects with join points and advices) are generated and integrated into the resulting system without further intervention. At runtime, interactive visualizations enable efficient tracing between log file entries and corresponding model artefacts.

If logging output is constantly observed, the models created at design time turn into runtime models which contribute to the overall understanding of the architecture while searching for anomalies. Changes to these models can be propagated to the running system to gain instant feedback if anomalies have been resolved. Our current prototype allows feedback to both structural and behavioural models, but is not yet able to propagate changes back to the running system. We plan to add this feature in the future, but we also see more potential in the idea of Linking and Brushing to reveal interrelationships between generic data sources and runtime models. For example, a time series interval of performance data could be selected which highlights model elements responsible for the selected data, ideally sorted by relative distribution (e.g., method *sort* produced 65% of the selected load) and accessible by selectable granularity (e.g., packages, classes and methods). Bottlenecks could be identified, repaired and changes be propagated to the running system. Interactive visualizations thus can be seen as abstraction/aggregation layer between running systems and their causally connected runtime models. They augment conventional runtime model, debugging

and tracing approaches and help developers to understand the system, reproduce errors and find the root cause of problems in models created at design time.

Nevertheless, the initial effort must not be underestimated since the code generator must be implemented and produce proper output to see first results. Furthermore, the overall performance of the system has yet to be tested since a large number of model elements and log file entries may result in increased effort regarding code generation, log file analysis, repository lookup and interactive data presentation. This is especially the case when using techniques like Linking and Brushing where multiple models and analysis views must be traversed to highlight particular model elements. Further work is required to see if these concerns are reasonable and if so, to find out how these problems can be solved.

6 Related Work

Maoz [9] introduces a similar approach which uses model-based traces to record the runtime of a system through abstractions provided by models used in its design. The traces include information about the systems from which they were generated, the models that induced them and the relationships between them at runtime. The approach focusses on metrics and operators for such traces to understand the structure and behaviour of a running system while our approach focusses on interactive visualizations which serve as abstraction and aggregation layer between the running system and its runtime models. An overview of trace exploration tools and techniques is given by Hamou-Lhadj and Lethbridge [10].

Graf and Müller-Glaser [4] tackle the problem of identifying error occurrences by defining various debugging-perspectives independent of the actual execution-platform. They extract runtime information out of the executed binary from the target platform and transport this information back to the model level. Obtained runtime information can then be used to visualize the internal state of the executable. In contrast to our approach, they extend the UML meta-model with meta-classes that allow storage of data acquired by the model mapping.

Like in our approach, Fuentes and Sanchez [11] also use aspect-oriented techniques to cope with error occurrences. They implemented a dynamic model weaver to run aspect-oriented models where aspects are woven and unwoven during model execution, leading to reconfiguration of the system by well-defined adaptation points. Interactive model simulation is achieved by loading and unloading pointcuts that manipulate aspects. Compared with our approach, their weaver focusses on testing and debugging a model itself before moving into an implementation while we concentrate on root cause analysis of running systems.

An approach which also uses log files to coordinate model operations is introduced by Bodenstaff et al. [12]. In their approach, the event log is assumed to be consistent with the running system and with a model if no contradictions with the model exist. The challenge lies in identifying relevant parts of the log and abstracting them to enable further processing. They suggest to either adapt the system to produce log files in a proper format or to reconstruct data from raw event logs. Compared to our approach, the output format of the log files can

directly be manipulated within the code generator. Furthermore, their goal is to perform consistency checks whereas in our work locating errors and unwanted behaviour through interactive visualizations is of particular interest.

Regarding model access, Holmes et al. [3] also use a model-aware repository in combination with unique identifiers. The repository can be queried through web services and is able to store multiple versions of models. These models describe a service-based system and its requirements and are used at runtime to trace back violations. Their goal is to check compliance to regulations whereas our work focusses on improving root cause analysis with interactive visualizations.

Dettmer [5] defines *root cause* with the help of Current Reality Trees (CRT). Such trees contain chains of undesirable effects where a root cause is a special undesirable effect without a predecessor (that is, a leaf of the CRT). Such trees ease the analysis of problems since they enable explicit modelling of dependencies between undesirable effects. It can be considered as visualization technique without interactive aspects, but the idea could be adapted in a way so that call hierarchies are navigable to the last occurred exception or undesired software behaviour. Elgammal et. al. [13] also make use of CRTs for root cause analysis to resolve compliance violations in service-oriented environments. In contrast to our work, their approach focusses on resolving design time violations.

Root cause analysis with the presence of huge amount of data during process execution is also discussed by Rodriguez et al. [14]. Their approach addresses conformance to rules and the understanding of non-compliance in service-based business processes. In contrast to our approach, their solution lies in reporting dashboards and decision trees to encounter the huge amount of data while our work focusses on the usage of runtime models with highest possible reusability.

7 Conclusions and Future Work

In this paper we combined model-driven techniques with runtime models to augment root cause analysis of executing systems. We gave an overview of our approach to illustrate the development steps which cover software models created at design time and their runtime utilization when analyzing file output produced by the system under observation. We described our model-driven strategy which uses a repository to provide runtime access to model-relevant data for the system or other external applications. We introduced interactive visualizations of which we implemented a list view, a matrix view, an outline view and a call matrix view in our prototype. We analyzed advantages of our approach as well as potential performance drawbacks if models tend to increase in complexity.

We plan to perform an empirical evaluation of our approach to analyse its applicability. Furthermore, we aim to generalize our prototype to accept more meta-models than UML by adapting the code generator and to include additional runtime information sources like network traffic or workload of individual software parts. We especially see potential in improving the code generator to produce additional output, e.g. automated reports in case of unexpected behaviour. We will compare our work with existing transformation frameworks which pro-

vide basic tracing features and consider including code-to-model transformations to support existing implementations. Future versions will include better controlling of the log file output and propagation of changes to the running system.

References

1. Bencomo, N.: On the use of software models during software execution. In: Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering. MISE '09, Washington, DC, USA, IEEE Computer Society (2009) 62–67
2. Vogel, T., Neumann, S., Hildebrandt, S., Giese, H., Becker, B.: Model-driven architectural monitoring and adaptation for autonomic systems. In: Proceedings of the 6th international conference on Autonomic computing. ICAC '09, New York, NY, USA, ACM (2009) 67–68
3. Holmes, T., Zdun, U., Daniel, F., Dustdar, S.: Monitoring and analyzing service-based internet systems through a model-aware service environment. In: Proceedings of the 22nd international conference on Advanced information systems engineering. CAiSE'10, Berlin, Heidelberg, Springer-Verlag (2010) 98–112
4. Graf, P., Müller-Glaser, K.D.: Dynamic mapping of runtime information models for debugging embedded software. In: Proceedings of the Seventeenth IEEE International Workshop on Rapid System Prototyping. RSP '06, Washington, DC, USA, IEEE Computer Society (2006) 3–9
5. Dettmer, H.W.: Goldratt's theory of constraints: a systems approach to continuous improvement. ASQ Press (1997)
6. Van Ham, F.: Using multilevel call matrices in large software projects. In: Proceedings of the Ninth annual IEEE conference on Information visualization. INFOVIS'03, Washington, DC, USA, IEEE Computer Society (2003) 227–232
7. Rao, R., Card, S.K.: The table lens: merging graphical and symbolic representations in an interactive focus+context visualization for tabular information. In: Conference Companion on Human Factors in Computing Systems. CHI '94, New York, NY, USA, ACM (1994) 222–
8. Keim, D.A.: Information visualization and visual data mining. IEEE Transactions on Visualization and Computer Graphics **8**(1) (January 2002) 1–8
9. Maoz, S.: Using model-based traces as runtime models. Computer **42**(10) (October 2009) 28–36
10. Hamou-Lhadj, A., Lethbridge, T.C.: A survey of trace exploration tools and techniques. In: Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research. CASCON '04, IBM Press (2004) 42–55
11. Fuentes, L., Sánchez, P.: Transactions on aspect-oriented software development vi. Springer-Verlag, Berlin, Heidelberg (2009) 1–38
12. Bodenstaff, L., Wombacher, A., Reichert, M., Wieringa, R.: Made4ic: an abstract method for managing model dependencies in inter-organizational cooperations. Serv. Oriented Comput. Appl. **4**(3) (September 2010) 203–228
13. Elgammal, A., Türetken, O., van den Heuvel, W.J., Papazoglou, M.P.: Root-cause analysis of design-time compliance violations on the basis of property patterns. In Maglio, P.P., Weske, M., Yang, J., Fantinato, M., eds.: ICSOC. Volume 6470 of Lecture Notes in Computer Science. (2010) 17–31
14. Rodríguez, C., Silveira, P., Daniel, F., Casati, F.: Analyzing compliance of service-based business processes for root-cause analysis and prediction. In: Proceedings of the 10th international conference on Current trends in web engineering. ICWE'10, Berlin, Heidelberg, Springer-Verlag (2010) 277–288