6th International Conference on Computer and Knowledge Engineering (ICCKE 2016), October 13-14 2016, Ferdowsi University of Mashhad

Towards a Tracing Framework for Model-Driven Software Systems

Fazilat Hojaji
Department of Software Engineering
University of Isfahan
Isfahan, Iran
f.hojaji@eng.ui.ac.ir

Bahman Zamani
Department of Software Engineering
University of Isfahan
Isfahan, Iran
zamani@eng.ui.ac.ir

Abdelwahab Hamou-Lhadj
Department of Electrical and Computer Engineering
Concordia University
Montréal, Québec, Canada
wahab.hamou-lhadj@concordia.ca

***Abstract*—Understanding software behavior by analyzing its execution traces is an important enabler for many software engineering tasks. In Model-Driven Development (MDD), dynamic analysis methods are often used to analyze executable models to enable the understanding of software behavior in early stages of the development process. An execution trace of a model can provide information to help reason about executable models. However, understanding an execution trace is not an easy task due to the size and complexity of typical traces. In this work, we aim at tackling this problem by proposing a model tracing framework, comprising compaction techniques to simplify the analysis of large traces at a higher level of abstraction, and a model tracing language, to capture run-time behavior of the executed model adequately.**

Keywords—Dynamic Analysis; Model Tracing; Executable Model; Model Driven Development

I. INTRODUCTION

Model-Driven Development (MDD) is a software development paradigm that aims to decrease the complexity of software systems by raising the level of abstraction [18]. In this paradigm, models are the key artifacts in the software development process. The success of the development process is mainly related to the quality of models [15]. Hence, appropriate techniques are required for ensuring the quality of models. Some researchers (e.g., [5]) have developed static analysis techniques that are based on well-defined models of behavior to verify the correctness of models by assessing their static properties. Another alternative is to apply dynamic analysis methods to verify the quality of models via executing them [18]. Dynamic analysis typically involves the analysis of execution traces that are generated from executing the models. An execution trace provides the information necessary to analyze the run-time behavior of an executable model [19]. However, traces tend to be very large and difficult to analyze. There is a need for techniques to reduce large traces while keeping their main essence. Coping with the large size of typical traces is a challenging task in both code-centric and model-driven development [9]. Existing trace compaction approaches apply specific visualization and abstraction techniques to overcome the vast size of execution traces [9]. However, these techniques have only been applied to traditional systems. The effectiveness of them has yet to be shown when applying to executable models.

The growing complexity of dynamic analysis is the motivation behind work on trace compaction. In particular, current research in the area of model driven engineering (MDE) is primarily concerned with reducing the size of traces and simplifying understanding software behavior through the use of compaction techniques.

Our research deals with executing and tracing of UML activity diagrams. We focus on executing activity diagrams because one major principle of UML's semantics is that every behavior in a system is eventually modeled by actions and hence, each behavior in UML is expressed using a set of actions [22]. The fUML standard defines the execution semantics to implement a model interpreter for UML activity diagrams or more precisely for fUML activity diagrams [17].

It is currently unclear how the run-time information of executable UML models can be obtained in a compact form. The main objective of this research is to investigate tracing and monitoring techniques for model-driven systems. More precisely, we aim to develop a model tracing framework which includes 1) techniques for facilitating the analysis and understanding of large traces that are produced from executing UML models; 2) a trace language to represent traces in a compact form and make model tracing more uniform and standard. The techniques that we propose will provide the ability to measure various run-time properties of models that should enable various software maintenance and program comprehension tasks.

In this research, we apply the problem-centered approach of the design science research methodology (DSRM) presented by Peffers et al. [26], while aligning the seven guidelines for design science, defined by Hevner et al. [10], to develop the model tracing framework. The process starts with identifying common properties of existing model tracing approaches and studying the techniques implemented in trace analysis tools through the study of the literature. In the next step, a set of simple and practical metrics to measure various properties of execution traces of UML models are proposed. According to these metrics, a set of trace abstraction methods tailored to model-driven systems are developed, which can help software engineers comprehend and analyze the behavior of models at various stages of the development process. By using the

results of two previous steps, a trace metamodel for fUML is developed to capture the execution of UML models in a compact representation form. Finally, we use common techniques of evaluation in design science research [10] and examine the proposed language capabilities and the compaction techniques by carrying out empirical studies and present the quantitative as well as the qualitative results of the case studies.

The paper is organized as follows: Section 2 explains executable UML and describes the principles for executing UML models. Section 3 outlines related works. Section 4 motivates this work by presenting an example that highlights the need for reducing size of model traces. In Section 5 we introduce a prototype for the analysis and visualization of the traces generated from UML models.. The paper ends with the conclusion and future works in Section 6.

II. EXECUTABLE UML

This section describes briefly how executable models in UML can be constructed. Two main elements are necessary to make a modeling language executable [7]: (1) an action language, to specify the behavioral semantics of a modeling language; and (2) an operational semantics to specify where and how the actions can be placed in a model and how a model must be interpreted. In the first step of the process of building a UML executable model, the structural aspects of the model is established by means of class diagrams. After that, the behavior of the operations (sets of actions) of the defined classes is defined using an action language. An action language is used for defining operational semantics. Kermeta [20] and fUML [23] are examples of action languages. The fUML subset consists of UML modeling concepts for defining the structure of a system with UML classes and the behavior of UML classes with UML activities. Using fUML action language, the full behavior of UML models can be specified using a set of atomic actions [23].

III. RELATED WORKS

The basis of our work is directly related to the field of dynamic analysis, especially in the context of visualization of run-time information [2] and trace compaction techniques [10, 9]. In this section, we briefly present the closest related approaches that have been conducted in domain of code-centric and model driven development.

1.1. Code-centric approaches

In the field of code-centric development, many compression and compaction approaches have been proposed for tackling the large volume of traces. We classified our identified techniques into several different groups:

1.1.1. Visualizing or Exploring an Execution Trace

Trace visualization, as in the approaches of Prada-Rojas et al. [27] and Kunihiro et al. [14], is the extraction of high level views of the run-time information to support system comprehension. Most approaches (i.e. [9, 16, 24]) use a UML sequence diagram to visualize interactions among grouped objects and depict the behavior of the program. Takechi et al. [13] proposed a technique to generate abstracted sequence diagrams based on the information of applied GoF design patterns in a source code. They defined some grouping rules for each design patterns and then, grouped objects in execution trace based on information of applied design patterns to abstract execution trace. Sharp et al. [28] proposed several methods to explore a large-scale sequence diagram. They applied some methods (e.g. filtering methods based on “the time” and zooming function) for reducing the amount of run-time information.

1.1.2. Abstracting the History of Object Interactions

Hamou-Lhadj et al. [9] proposed a method for obtaining the summary of an execution trace by removing utility objects which do not implement key system concepts. They proposed a metmodel called the CTF (Compact Trace Format) which represents traces of routines calls as directed acyclic graphs. This way, common subtrees are represented only once. They showed that this native compaction can result in almost 90% compaction ratio. Taniguchi et al. [30] proposed a method to extract compact sequence diagrams by abstracting some repetition patterns and recursive calls appearing in the trace. In this approach, a repetition of similar sub-trees in a call tree are detected and replaced with one representative, which shows whole repeated structure and the number of repetitions. Myers et al. [21] followed the similar approach to reduce the size of sequence diagrams for industrial software through compacting loop iterations.

1.1.3. Extracting Behavior from an Execution Trace

Quante and Koschke [11] introduced a technique to build an object process graph through dynamic analysis using run-time information. In this approach, the behavior of different components of a program is extracted as dynamic object process graphs. The size of graph can be reduced by removing branch nodes, unnecessary label nodes, local loops and irrelevant subgraphs. These are repeatedly applied until the graph cannot be simplified any more. Noda et al. [12] proposed a method to slice the reverse-engineered sequence diagram. The method extracts and visualizes the parts of trace that a developer can focus on them.

1.1.4. Partitioning and clustering

Dugerdil and Repond [4] used a software clustering technique based on the dynamic analysis of method calls while executing a scenario of a system. They implemented a clustering technique for identifying the set of functional components by splitting the execution trace in contiguous segments and observe collaborating classes presentet in each segment. Zaidman and Demeyer [31] proposed a heuristic approach to manage the volume of the trace by searching for common global frequency patterns. They analyzed consecutive samples of the trace to identify recurring patterns of events, which have the same global frequencies and to find patterns that are more interesting and more frequent.

1.1.5. Dynamic slicing

Smith and Korel [29] proposed a technique of slicing event traces to reduce the number of events for analysis. This

technique uses a slicing algorithm to identify several types of dependencies between events. All events that are irrelevant or do not affect the starting event are removed from the event trace that can further reduce the size of the sliced event trace. Dhamdhere et al. [3] followed similar approach and provided a compact execution history for dynamic slicing of programs by focusing on critical statements in a program. An instrumentation algorithm is used to identify critical nodes and similar loops for the summarization. In this technique, only critical statements appear more than once in an execution trace and all other statements appear at most once.

1.2. Model-based approaches

In comparison to trace compaction in code-centric development, considerably less work has been devoted to model tracing and only a few model tracing approaches have been proposed in the literature. Most important approaches are described in the following. However, trace compaction and summarization has not been considered in these approaches completely.

Maoz [16] proposed a model-based trace approach, which captures the run-time behavior of a system at the abstraction level of the design models. This approach allows following and monitoring system design models, such as sequence diagrams, class diagrams, and state charts during model execution. Aljamaan and Lethbridge [1, 8] applied a different approach to enable model level tracing. They proposed Umple[1] -an action language in a fully executable platform- for experimenting and developing action languages. They defined trace directives that allow modelers to specify traces of UML attributes and state machines at the model level. Fuentes et al. [6] provided Populo to represent trace records in the tree structure which traces are rendered visually. A tree represents execution of the activities and called sub-activities. For each action, some information about the inputs and outputs is also displayed. Populo has the ability to customize the amount of information by the end-user who decides how many details he/she needs at any time. Mayerhofer [18] proposed an approach to capture execution traces of fUML models and provides the basis for analyzing of fUML models. She defined a metamodel containing operational semantics for execution traces that can be used for creating trace models. Despite the special capabilities of this approach, there is no compaction for execution traces and the scalability in terms of trace length and model size can be challenging and expensive. Recently, Bourse et al. [2] presented a generative approach to automatically derive a domain-specific trace metamodel for an executable domain-specific language by analyzing its definitions of execution states and events. In this approach, execution states contain the values of the mutable properties of a model. Thereby, the trace metamodel provides scalability in time and reduces redundancy when manipulating traces.

Overall, a limited number of research papers were found that directly relate to model trace compaction. This indicates that this area needs further investigation.

[1] http://cruise.eecs.uottawa.ca/umple

IV. MOTIVATION

To illustrate the issues of modeling and tracing an fUML model, we consider in this section a simple example shown in Figure 1. This model defines two classes University and Student which have two attributes. Student class has firstname and university class has name attribute. The class Student defines NewStudent() and the class University defines addStudent(), whose behaviors are defined by the depicted activities. The operation NewStudent() calls the operation addStudent(), which creates a new Student object and adds the created student to the collection of students enrolled at the university. The modeling can be done by using an Eclipse plug-in based on EMF. For executing these models and capturing the respective execution traces, we used the fUML execution prototypes proposed by Mayerhofer [18] that provides an event model, a command API, and a dedicated trace model. The trace is created as soon as the execution of the first activity (depicted in part b of figure 1) starts and it is updated on each state change of the execution. In this example, the run-time behavior of executed models is captured according to the defined trace model. However, such metamodel cannot create traces in a compact form. Hence, the generated traces can be arbitrarily large. For instance, more than 135 records of xml-based traces that have an average length of 3000 bytes are generated from the execution of the activity example of figure 1. The same activity might be executed several times during the execution of an fUML model. Therefore, the size of execution traces can be increased respectively. A small part of the textual trace is shown in figure 2 to demonstrate that it is difficult to read. The lines of the traces have different size that contain information about executed model elements (e.g. activities, actions, and control nodes). Also, inputs and outputs of activities as well as the chronological execution order and logical order of model elements are captured. Each line of the trace has a containment reference to corresponding model elements that are identified by object ID in related UML file. Because of large amount of trace, we have exported the trace data to a database and implemented an analysis tool prototype for analyzing the resulting view. In the next section, we briefly present this prototype, point out which capabilities have been applied for analyzing the trace, and discuss the ideas to compact the trace that have not been implemented in the prototype yet.

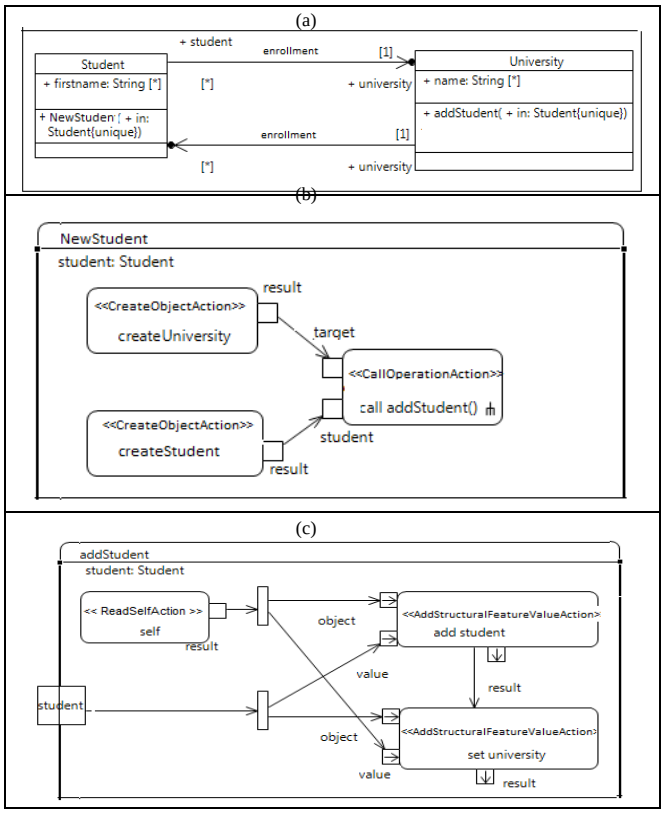

Figure 1. Model execution example: Excerpts of fUML models

Figure 2. Excerpt of the trace of example fUML model

V. TRACE VISUALIZATION AND TRACE ANALYSIS

We implemented a prototype for exploring large execution traces of fUML models. It takes traces of fUML models and corresponding UML files in xml format as input and displays the trace using visualization techniques. This prototype was implemented as an Eclipse plug-in based on the Eclipse Modeling Framework. We can collapse and expand parts of the traces and analyze the execution traces of the example traces. Figure 3 shows a screenshot of our prototype. It allows designers to visualize the behavior of an fUML model by interpreting the UML actions. The traces are visualized in a condensed tree-like form. The tree has hierarchically grouped information to represent the entire trace. The tree levels correspond to the call hierarchy among executed activities, the nesting of activity node executions as well as run-time values of objects. This view lets the user browse the trace at various levels of detail. The root node of the tree is the model name. Its children are nodes representing activities (operations). An activity node contains node executions, which have nodes representing their actions, types, inputs and outputs. Each node presents the element name extracted from the fUML model and the respective type of model element. The name of each node has been obtained from corresponding UML element. Prior to the execution of an activity, we have to manually determine input values for global variables defined in the model. After executing actions, the run-time instances of objects are assigned to attributes. These run-time instances are displayed in traces as run-time values and grouped by featurevalues, feature and type.

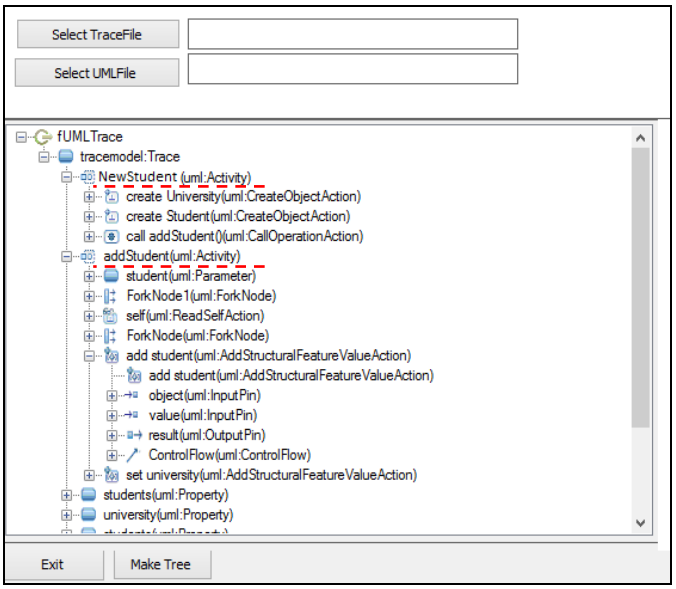

Figure 3. Excerpt of the trace of the example fUML model

Despite the obvious advantages of visualization, the analysis and understanding of our sample trace is still difficult. Hence, we need to hide some components of the trace in order to abstract out its content. To help the user extract useful information, we have applied several filtering techniques, which are described as follows:

4.1. Nesting of activity node executions compression: This technique consists of collapsing the internal invocations that derive from a given call. A click on this call will result in making its internal invocations visible. For example, the

analyst may simply decide to hide all the invocations of a specific action. In Figure 3, NewStudent and addStudent are related to the execution of activities carried out during the execution of our example model. Each activity has been collapsed which can be expanded by a click from the user. These activities include the actions constituting the events issued during the execution of the activities NewStudent and addStudent (depicted by thin dashed red lines) for the defined context object and input parameter values.

4.2. Pruning: Pruning is the process of removing information from the tree in order to reduce its size [25].We implemented two kinds of pruning techniques: Action Execution pruning and RuntimeValues pruning. Pruning an action execution consists of removing its corresponding actions from subtrees of activities. According to [23], fUML defines operations that support the manipulation of objects and the logical constructors. Examples of these actions are object creation, calls to methods or writing an attribute value, among others. Hence, all subtrees of the trace with the type of activity execution are removed in the pruning process. CreateObjectAction, DestroyObjectAction, ReadSelfAction, ReadStructuralFeatureAction are some kinds of action types that are removed.

Runtimevalues pruning consists of performing the same task on runetimevalues. The subsequent featurevalues that derive from the runtimevalues are removed. After implementing the pruning techniques in the prototype and applying it to our example traces, the average size of the resulting traces was 33 rows, which is around 60% and still too high for large models.

4.3. Pattern matching: Pattern matching is a mechanism to detect similar sequences of events in the form of execution patterns [25]. The preliminary results of analyzing the example trace showed the fUML trace consists of recurring execution patterns such as invoking operations and assigning their inputs and outputs. However, we know that exact matching is too restrictive and does not reduce the size of traces very much. Therefore, we have to use inexact matching. The differences of patterns are mostly in identity of model elements, order of node executions and loops repetition. A set of matching criteria are required to determine which two sequences of events can be considered equivalent. A preliminary set of criteria include the identity of model elements, the certain depth of the trace tree as well as the number of repetitions in loops. Note that the different combinations of matching criteria will result in different filtering of the content of trace and should be evaluated in practice.

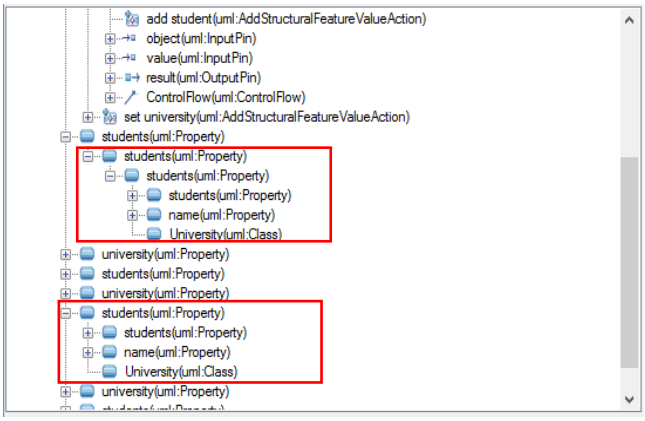

Figure 4. Repetition of execution patterns in the trace

Figure 4 shows repetition of execution patterns about the created Student object. These two Student objects constitute two distinct snapshots of the same run-time object. The first snapshot captures the state of this object after its instantiation and the second snapshot after the initialization of the firstname attribute. If we expand the tree nodes in second level, we would find that the collapsed portions display the same patterns. While, our example has repetitions only in run-time values of Student and University objects; often repetition rates are much higher especially in activity executions. Moreover, loops can be nested in activity diagrams, and represented as repetitive patterns at any level of nesting. We are currently working on implementing of generalization mechanism to identify repetitive execution patterns and consider one instance of the same patterns. Generalization mechanism [9, 25] has been used as a trace compaction technique in code-centric approaches. We are motivated to investigate how this technique can be used to summarize fUML execution trace.

VI. CONCLUSION

Analyzing and reasoning about execution traces is a challenge due to the large size of typical traces. In order to analyze large amount of execution traces, an efficient representation of trace information is required. Trace compaction and abstraction techniques are also needed to provide traces with a compact view of a set of execution traces. In this paper, we presented ideas and requirements for addressing these issues that we packaged in the form of a model tracing framework. In our pilot experiment, we evaluated some techniques to explore and filter the trace content and make the analysis of model traces considerably easier. Results showed that using filtering techniques was more efficient to the analysis of model traces. Future work involves the development of pattern matching to overcome the trace size explosion problem. Another direction of future work is the development of an expressive trace language to capture various UML elements at run-time in a compact representation form.

REFERENCE

[1] H. Aljamaan, T.C. Lethbridge, O. Badreddin, G. Guest, A. Forward, Specifying Trace Directives for UML Attributes and State Machines, 2nd International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2014, IEEE, Lisbon, Portugal 2014, pp. 79 - 86.

[2] E. Bousse, T. Mayerhofer, B. Combemale, B. Baudry, A Generative Approach to Define Rich Domain-Specific Trace Metamodels, 11th European Conference on Modelling Foundations and Applications (ECMFA), 2015.

[3] D.M. Dhamdhere, K. Gururaja, P.G. Ganu, A compact execution history for dynamic slicing, Information Processing Letters 85 (2003) 145-152.

[4] P. Dugerdil, J. Repond, Automatic generation of abstract views for legacy software comprehension, Proceedings of the 3rd India software engineering conference, ACM, 2010, pp. 23-32.

[5] R.B. France, B. Rumpe, Model-Driven Development of Complex Software: A Research Roadmap, Future of Software Engineering (FOSE 07), 2007, pp. 37-54.

[6] L. Fuentes, J. Manrique, P. Sanchez, Execution and simulation of (proled) UML models using Populo, Proceedings of the 2008 international workshop on Models in software engineering., 2008.

[7] L. Fuentes, P. S´anchez, Dynamic Weaving of Aspect-Oriented Executable UML Models, Transactions on AOSD VI, LNCS 5560, pp. 1–38, 2009.

[8] H.Aljamaan, T.C. Lethbridge, Towards Tracing at the Model Level, 19th Working Conference on Reverse Engineering, IEEE, Kingston, ON 2012, pp. 495-498.

[9] A. Hamou-Lhadj, Techniques to Simplify the Analysis of Execution Traces for Program Comprehension, Ottawa-Carleton Institute for Computer Science School of Information Technology and Engineering University of Ottawa,Ottawa Ontario Canada, University of Ottawa, 2005, pp. 171.

[10] A.R. Hevner, S.T. March, J. Park, S. Ram, Design science in information systems research, MIS Quarterly 28 (2004) 75–105.

[11] J. Quante, R. Koschke, Dynamic Object Process Graphs, Journal of Systems and Software 81 (2008) 481-501.

[12] T.K. K. Noda, K. Agusa, and S. Yamamoto, "Sequence Diagram Slicing," in APSEC '09, 2009, pp. 291–298.

[13] T.T.T. Kobayashi, N. Atsumi, K. Agusa, Grouping Objects for Execution Trace Analysis Based on Design Patterns, Software Engineering Conference (APSEC), IEEE, 2013.

[14] N. Kunihiro, T. Kobayashi, K. Agusa, Execution trace abstraction based on meta patterns usage., Working Conference on Reverse Engineering (WCRE), IEEE, 2012 pp. 167-176.

[15] Y. Laurent, R. Bendraou, M. Gervais, Executing and Debugging UML Models: an fUML extension, Proceedings of the 28th Annual ACM Symposium on Applied Computing ACM, 2013, pp. 1095-1102.

[16] S. Maoz, Using Model-Based Traces as Runtime Models, IEEE Computer Society (2009) 28-36.

[17] T. Mayerhofer, Breathing New Life into Models An Interpreter-Based Approach for Executing UML Models, The Vienna University of Technology, Faculty of Informatics, 2011, pp. 103.

[18] T. Mayerhofer, Defining Executable Modeling Languages with fUML, Vienna University of Technology, Austria, Vienna, 2014, pp. 252.

[19] T. Mayerhofer, P. Langer, G. Kappel, A Runtime Model for fUML, MRT '12 Proceedings of the 7th Workshop on Models@run.time ACM, 2012, pp. 53-58.

[20] P. Muller, F. Fleurey, J. Jézéquel, Weaving Executability into Object-Oriented Meta-Languages, MODELS2005, Springer Berlin Heidelberg, 2005, pp. 264-278.

[21] D. Myers, M.-A. Storey, M. Salois, Utilizing debug information to compact loops in large program traces, Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on, IEEE, 2010, pp. 41-50.

[22] OMG, UML Super structure, 2011.

[23] OMG, Semantics of a Foundational Subset for Executable UML Models (fUML), Object Management Group, 2013.

[24] W.D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, J. Yang, Visualizing the Execution of Java programs, Proceedings of the International Seminar on Software Visualization, Springer-Verlag, 2002, pp. 151-162.

[25] W.D. Pauw, D. Lorenz, J. Vlissides, M. Wegman, Execution Patterns in Object-Oriented Visualization, Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems, 1998, pp. 219-234.

[26] K. Peffers, T. Tuuanen, M.A. Rothenberger, S. Chatterjee, A Design Science Research Methodology for Information Systems Research, Journal of Management Information Systems (2007) 45-77.

[27] C. Prada-Rojas, M. Santana, S. De-Paoli, X.Raynaud, Summarizing embedded execution traces through a compact view, Conference on System Software, SoC and Silicon Debug S4D, 2010.

[28] R. Sharp, A. Rountev, Interactive exploration of UML sequence diagrams, Visualizing Software for Understanding and Analysis, 2005. VISSOFT 2005. 3rd IEEE International Workshop on, IEEE, 2005, pp. 1-6.

[29] R. Smith, B. Korel, Slicing event traces of large software systems, arXiv preprint cs/0101005 (2001).

[30] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, K. Inoue, Extracting sequence diagram from execution trace of Java program, Principles of Software Evolution, Eighth International Workshop on, IEEE, 2005, pp. 148-151.

[31] A. Zaidman, S. Demeyer, Managing trace data volume through a heuristical clustering process based on event execution frequency, Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on, IEEE, 2004, pp. 329-338.