Application of Tracing Techniques in Model-Driven Performance Engineering

Mathias Fritzsche[1], Jendrik Johannes[2], Steffen Zschaler[2], Anatoly Zherebtsov[3], and Alexander Terekhov[3]

[1] SAP Research CEC Belfast & Queen's University Belfast
United Kingdom
`mathias.fritzsche@sap.com` & `mfritzsche01@qub.cu.uk`

[2] Technische Universität Dresden
D-01062, Dresden, Germany
`jendrik.johannes@tu-dresden.de, steffen.zschaler@tu-dresden.de`

[3] XJ Technologies Company
195220 St. Petersburg , Russian Federation
`anatoly@xjtek.com, talex@xjtek.com`

Abstract. In our previous work we proposed Model-Driven Performance Engineering (MDPE) as a methodology to integrate performance engineering into the model-driven engineering process. MDPE enables domain experts, who generally lack performance expertise, to profit from performance engineering by automating the performance analysis process using model transformations. A crucial part of this automated process is to give performance prediction feedback not based on internal models, but on models the domain experts understand. Hence, a mechanism is required to annotate analysis results back into the original models provided by the domain experts. This paper discusses various existing traceability methodologies and describes their application and extension for MDPE by taking its specific needs into account.

1 Introduction

Model-Driven Engineering (MDE) is a technique for dealing with the ever-increasing complexity of modern software systems. It places models of software—often expressed using domain-specific languages (DSLs)—at the heart of the development process. This enables developers to view and design a software system from a much higher level of abstraction than the code level, allowing them to cope with much higher levels of complexity. Additionally, the use of DSLs allows domain experts to be involved in the development of a software systems. This can increase the quality of software as the domain requirements can be taken into account more directly and accurately. For example, the authors of [1] describe how DSLs can be used to develop so called Composite Applications that access services provided by a SAP Business Suite system. This is supported by industrial tools, such as the Composition Environment (CE) [2] that applies MDE

for the development of Composite Applications, enabling experts in a domain to build new applications based on pre-provided modules.

However, a difficulty with MDE lies in supporting extra-functional requirements in the software system. As generic solutions that guarantee certain non-functional properties under any circumstance are typically difficult to provide, developers need expertise regarding specific non-functional properties and how to support them in application design. This expertise is typically lacking with domain experts. Additionally, the high-level of abstraction beneficial to the development of complex applications, can also make it difficult to provide reasonable estimates for non-functional properties of the resulting system.

Therefore, there is a need for a better support for non-functional properties within MDE. Performance is one such important property, which has been researched in the context of MDE [3, 4] and is also the focus of this paper. We have previously proposed Model-Driven Performance Engineering (MDPE) [5], an extension of MDE that allows performance analysis models to be derived from development models at each level of abstraction. However, so far, the results of such an analysis still require performance engineering expertise to be interpreted. In particular, the performance engineer must understand the specific analysis or simulation technique used and be able to translate back the results from this analysis into properties of the original development models. In this paper, we investigate how this feedback of results can be automised in the context of MDPE, such that domain experts can benefit from analysis results without consulting performance engineers.

Trace information about all of the various transformations that together make up MDPE is the most important asset required for implementing result feedback. Therefore, in this paper, we will discuss various approaches to collect and maintain such trace information. We will then discuss which of these techniques is most appropriate in the context of MDPE and show how we have applied it to implement result feedback for MDPE. The contribution of this paper is, therefore, twofold: a) It presents a technique for feeding performance analysis results back into original development models, and b) to the tracing community it presents a case of application of tracing and a discussion of the benefits and drawbacks of a number of tracing techniques in a specific application context.

The remainder of the paper is structured as follows: We begin in Section 2 with a brief overview of MDPE including a description of where tracing information is required. Then, in Section 3 we describe the implementation of the feedback mechanism and also discuss which tracing technique is most suitable for this purpose. Section 4 describes related work and Section 5 concludes the paper.

2   Background

Performance engineering is used in software development to meet performance requirements in the design of a software system. Applying performance engineering is, however, costly since it requires performance experts, who understand the

formalisms that performance analysis and simulation tools use, to be consolidated. For this reason, it is often neglected or only done in the very beginning of system design. Consequently, the performance is only measured on the running system—which often leads to redesigns and reimplementations of (parts of) the system.

In Model-Driven Engineering (MDE), software is designed stepwise, by refining models until the concrete implementation is realized. Model-Driven Performance Engineering (MDPE) [5] proposes to do performance engineering at each of these refinement steps to discover design flaws early in the development process. Furthermore, it proposes to use model-driven techniques to automate performance engineering itself. To this end, we propose a semi-automatic generation of performance models based on development models (e.g. UML models) using model transformations. To have a sufficient cost-benefit this is also a stepwise process: basic analysis can be done automatically on each refinement level while more detailed analysis requires manual input to the generated performance models and therefore more performance expertise. Thus, MDPE takes two orthogonal dimensions of refinement into account: One dimension to refine the performance models, and another dimension to refine the development models in a traditional MDE process.

To define a process independent of development and performance analysis formalisms, we use a tool-independent performance metamodel. Development models from the MDE process are transformed into an instance of this metamodel: a Tool-Independent Performance Model (TIPM). Such a TIPM can be transformed into different performance analysis models called Tool-Specific Performance Models (TSPMs). These models are then employed by specific performance analysis tools using the same performance view-point on the system as common data base.

The TIPM offers a solution that is independent of any specific performance modelling concept, such as layered queuing models [6], stochastic petri nets [7], etc. Hence, an MDPE user is able to compare the capabilities of several performance modelling concepts without undergoing the error prone and time consuming task of defining the interfaces to the development modelling language in use [8]. Additionally, MDPE is independent of the performance analysis tool actually used, such as AnyLogic, etc., which simplifies the industrial application of MDPE. Finally, as a result of the TIPM we are able to support multiple kinds of development models, such as UML models, but also proprietary models used within SAP for the purpose of business behaviour modelling.

In [5] we presented a transformation from UML models to AnyLogic simulation models. This paper concentrates on the opposite direction: the tracing of results, collected by running the simulation models, to the UML models. To support this, the TIPM metamodel contains the concepts of monitors that can be filled with analysis results.

An excerpt from the metamodel is shown in Figure 1. The left side of the figure defines concepts that hold information about the structure of the studied system. These concepts—*Scenario*, *Step*, *PathConnection*, *Resource* and refine-
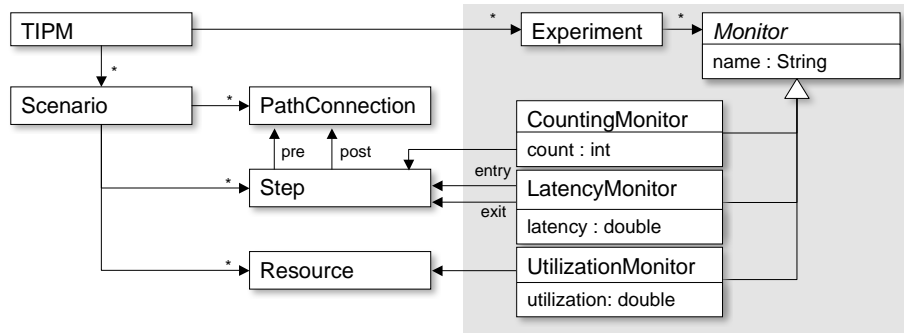
Fig. 1. The TIPM metamodel defines concepts to describe a system (left) and analysis results (right).

ments of those (not shown in the figure)—are based on the Core Scenario Model (CSM) introduced by Woodside et al. [9] and have basically the same semantics as defined there. The right side contains the concept of *Experiments* and *Monitors*, in addition to the concepts borrowed from the CSM. Those are used to indicate which kind of performance analysis should be performed and where. For the latter, different kinds of monitors can refer to different kinds of elements in a TIPM, which they observe. Their properties are only filled by the utilised analysis tool *after* an analysis has been performed.

In the figure, three different monitor types are defined: A *LatencyMonitor* holds information about the latency between two steps (*entry* and *exit*). A utilization measured for a resource can be placed into a *UtilizationMonitor*. A *CountingMonitor* observes how often a step is executed.

Like the transformation to a performance model, the tracing from a performance model is also a two-step process. In the first step, the simulation tool provides data to fill the monitors of a TIPM. Then this information can be used to update the development models from which that TIPM was generated. The following section discusses both steps in detail.

3 Extending MDPE with Traceability

Figure 2 provides an overview of our proposed architecture to extend MDPE with traceability. As shown in the figure, two steps, named as *Tool2TIPM Tracing* and *TIPM2DevelopmentModel Tracing*, are required in order to implement synchronization between performance analysis tools and development models. A description of both steps is provided in the following subsections.

3.1 Synchronization between Performance Analysis Tools and TIPMs

The tracing of simulation results back to a TIPM concentrates on filling the properties of monitor elements in the TIPM (cf. right side in Figure 1). These are
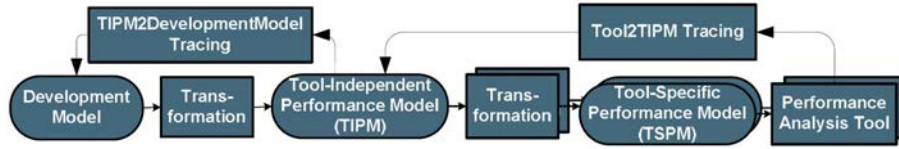
Fig. 2. Tracing architecture for MDPE as Block Diagram [10]

initially not set, because they are explicitly provided as containers for feedback information. The performance analysis tool, which is responsible for providing the result data, has to know about the monitors and their properties. This has to be taken into account, when transforming a TIPM into a TSPM.

As an example of such performance analysis tool we used AnyLogic developed by XJ Technologies [11]. It is a multi-method simulation tool, which includes basic services that can be used to create simulation models using different methods—*discrete-event*, *system dynamics* or *agent-based modeling*—and allows combining these methods in one model. The object-oriented model design paradigm supported by AnyLogic provides modular and incremental construction of large models. The simulation engine is based on Java technology, which makes it possible to use functionality provided by the Java runtime library in simulation models.

AnyLogic supports developing custom object libraries that can include model objects developed with the tool itself together with Java objects and third-party libraries written in Java. An AnyLogic library can be attached to a model development environment and its objects can be used in other simulation models. In order to support simulations based on TIPMs, we developed a special AnyLogic library that includes objects which behave corresponding to concepts from the TIPM metamodel and collect data about the simulated model during its executions. To generate AnyLogic simulation models, two transformations were developed using the Atlas Transformation Language (ATL) [12]. The first one converts a TIPM into a structure of AnyLogic library objects as anticipated by AnyLogic. It generates all required objects together with additional objects required to connect everything into a working model. This structure includes all AnyLogic objects and connections between them that have to be present in the model. The second transformation applies XML formatting to make the structure readable by the AnyLogic tool, effectively leaving the MDE technology space.

To enable the actual feedback, we have implemented a small service to which a simulation tool like AnyLogic can send information. In this way, the simulation tool itself does not require any MDE specific knowledge. It is sufficient to send a message to a designated port containing the information which TIPM (identified through its filename) and which property in which monitor to fill (addressed through their unique names). When receiving such a message, the service updates the corresponding TIPM with the provided information.

Monitors are also defined as objects in our AnyLogic library. Their main functionality is to collect the required result data during execution of the simulation model. Type and scope of the information collecting is defined in the TIPM: Parameters of monitors are transformed to parameters of library objects together with information on how to connect to the service listening for result data. As mentioned, the AnyLogic simulation engine can use a wide variety of features provided by Java; This was used to realise the connection to the service. After a model's execution, AnyLogic connects to the specified port and provides result data. Assuming, for instance, that AnyLogic has measured a latency of *11.38 ms* for a certain sequence of steps, it can set the *latency* value of the corresponding *LatencyMonitor* to `11.38`.

3.2   Synchronization between TIPM and Development Models

For the tracing between development models and TIPMs a solution is required to trace between two modeling languages where one, defined by the TIPM metamodel, is known, but the other, used for defining the development models, may vary. Our current MDPE prototype, however, only supports UML models as development models. In the future we require support for other (domain-specific) languages, such as SAP proprietary languages for business process modeling as shown in [1], as well.

In the following, different options to implement tracing between development models (of arbitrary metamodels) and TIPMs are analysed and one is selected. Afterwards, we exemplify the actual feedback process on UML development models using the chosen traceability methodology.

A straightforward option is the usage of bi-directional transformations, such as provided by the Query View Transformations (QVT) [13] relations language. Initial tool support has been published in [14]. We claim that this solution is tracing by design—in the sense that there is no need to care about tracing after the implementation of a transformation. However, it is more effort to develop bi-directional transformations than uni-directional ones as the transformation developer always has to keep both directions in mind. Thus, bi-directional are not an option for MDPE because we do not want to complicate the development of transformations between development models and TIPMs.

As another option, the transformation developer can provide a definition of how tracing information between development models and TIPMs are established after the transformation was performed. The Epsilon Comparison Language (ECL) [15] enables comparison of models of arbitrary metamodels. Hence, a transformation developer could use ECL to write a comparison specification using ECL that identifies correspondences between a development model and a TIPM. The disadvantage of this approach is that it currently requires manually writing comparison rules for each single transformation or, in other words, for each type of development model that should be supported. An approach supporting the definition of ECL rules in parallel with defining the transformations, could significantly reduce reduce the indicated overhead but is not available at the moment.

The approach we claim as most useful for the MDPE process is based on Higher-Order Transformations as available and described by Jouault [16] for the Atlas Transformation Language (ATL) [12]. Higher-Order Transformations are transformations that are used to transform one transformation A to a new transformation $A*$. This approach is used in [16] to automatically extend rules within ATL transformations with code for additional information generation. This code creates a tracing model when the transformation is executed. This tracing model conforms to a traceability metamodel, which is defined in [16] by extending the Atlas Model Weaver (AMW) [17] metamodel. This approach is not *traceability by design* as there is the need to consider tracing after implementing a transformation. However, the additional effort is simply executing a Higher-Order Transformation which has only to be defined for each applied transformation language but not for each single transformation. Additionally, the application of the Higher-Order Transformation can be integrated in the transformation tooling.

In our implementation, we execute the transformation provided by [16] to extend our current UML2TIPM transformation with tracing capabilities. Figure 3 depicts how, for instance, the rule "DeviceObject" is extended with traceability model generation capabilities. Hence, if the extended UML2TIPM transformation is executed, not only a TIPM but also a tracing model is generated.
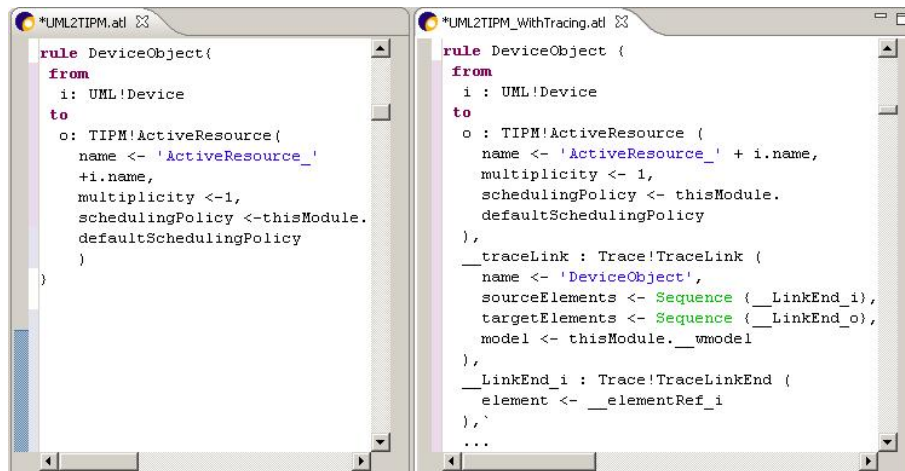

Fig. 3. Comparison between one ATL rule before (left) and after the HOT (right)

The tracing model enables us to annotate simulation results serialized by the monitor model elements in the TIPMs back to the original development models. Therefore, an Eclipse plug-in has been developed which iterates all monitors of a TIPM. Monitors are associated with Steps and Resources in the TIPM as described in subsection 3.1. It is the purpose of the plug-in to use the generated

tracing model in order to get the related development model element for the TIPM Step or TIPM Resource, the monitor is attached on.

For instance, if a *LatencyMonitor* is associated with the *Steps* "Initial_Node_1" (as entry) and "Final_Node_1" (as exit), the plug-in would analyze the trace links within the tracing model in order to get a reference to the UML Activity Diagram elements that were sources for the UML2TIPM transformation. In the case of the two *Steps* "Initial_Node_1" and "Final_Node_1" we get references to an *InitalNode* and a *FinalNode* in an UML Activity Diagram.

The actual annotation of the simulation result stored in the *LatencyMonitor*, which has been used as in the example in subsection 3.1, to the development models follows in a second step: The latency (*11.38 ms*) is annotated to the UML *Activity* containing the *InitalNode* and the *FinalNode* to which the trace links point. It has been mentioned that we do not only need to support UML models as development models but also other modeling languages such as proprietary languages used within SAP. A general solution for annotation is not possible since we have to take development language specifics, such as the mechanism used for the actual annotation of simulation results, into account. Therefore, we encapsulated the logic implementing the development language specific annotation of performance analysis results in one module, and the logic implementing the development language independent access of the TIPM and Tracing Model in another module.

In order to realize loose coupling between the modelling language specific part and the modelling language independent part, we used the standard extension mechanism provided by the Eclipse platform.

Thus, we implemented one Eclipse plug-in which implements the development language independent part of the TIPM to development model tracing, and provides an Eclipse Extension Point to be implemented by the development language specific Eclipse plug-in. We implemented such a plug-in in order to annotate the AnyLogic simulation results from the TIPM monitors back to UML models via the SPT profile [18].

By combining our transformation and tracing solutions, we created a prototype, which we successfully applied on examples. An issue often discussed when it comes to applying such transformation chains with tracing is information loss. For the application presented in this paper, however, we did not encounter any issues with transformation loss. Clearly, information is not preserved by our transformations; but that is intended, since only selected information is carried from development to transformation models and a different kind of information—the analysis results—are carried back.

4   Related Work

Our work has been strongly influenced both by needs arising from industrial practice and by previous work in the academic literature. Here, we briefly discuss some of the influences from the literature.

Grassi and Mirandola with their work in the area of component-based software-performance engineering were the first, in our knowledge, to present the notion of using model transformations in the MDA context for generating analysis models ([19], for example) for analysing performance. They propose refining a second line of models from the most generic models in parallel to those models meant for eventually generating executable code. In contrast, we propose to generate a new analysis model whenever it is needed, basing it on the most current development model available. Other authors have also proposed using model transformations for constructing analysis models. A more detailed discussion can be found in [5].

Our approach is much closer to work performed by Sabetta et al. [20], who present a new technique for transforming development models into analysis models using so-called abstraction-raising transformations. This work could be used as an extension of our work, although we would need to extend their specific transformation technique to support tracing in the way we need it.

Our TIPM metamodel is closely related to Woodside's work on CSM [9]. In fact, the TIPM metamodel is an extended version of CSM. Our main extension is the addition of the concept of monitors that enable us to indicate the specific performance properties of interest. As we have seen, these monitors play an important role in feeding information back into the development model as they will contain the analysis results. Based on the CSM, Woodside has gone on to build PUMA [8], a system quite similar to the work presented here. Feedback to development models is quoted as future work in [8], however.

5   Conclusion

We have presented the implementation of a performance analysis result feedback mechanism for MDPE based on Higher-Order Transformations for ATL. This technique helps developers to understand and experiment with performance effects of design decisions without the need for performance expertise. All that is required is the provision of basic performance annotations in the development models. In cases like the SAP case cited above, even this can be avoided by providing catalogues of available components already pre-annotated with correct performance data.

With the basic MDPE framework in place, we now need to perform experiments to support our claim that the result feedback is actually useful to domain experts. Such experiments will be performed in the experimentation phase of the MODELPLEX project and may lead to corresponding adjustments to MDPE. Also, in this context we will be implementing support for further input languages and simulation engines.

Acknowledgement

References

1. Fritzsche, M., Gilani, W., Fritzsche, C., Spence, I., Kilpatrick, P., Brown, J.: Towards utilizing Model-Driven Engineering of Composite Applications for Business Performance Analysis. In: ECMDA-FA'08 (to appear). (2008)
2. SAP AG: SAP NetWeaver Composition Environment 7.1 (2008) https://www.sdn.sap.com/irj/sdn/nw-ce/.
3. D'Ambrogio, A.: A model transformation framework for the automated building of performance models from UML models. In: WOSP '05, ACM (2005) 75–86
4. Gu, G.P., Petriu, D.C.: XSLT transformation from UML models to LQN performance models. In: WOSP '02, ACM (2002) 227–234
5. Fritzsche, M., Johannes, J.: Putting Performance Engineering into Model-Driven Engineering: Model-Driven Performance Engineering. In: MoDELS'2005 Satellite Events: Revised Selected Papers, LNCS 5002, Springer (2007)
6. Franks, R.G.: Performance analysis of distributed server systems. PhD thesis (2000) Adviser-C. Murray Woodside.
7. López-Grao, J.P., Merseguer, J., Campos, J.: From UML activity diagrams to Stochastic Petri nets: application to software performance engineering
8. Woodside, M., Petriu, D.C., Petriu, D.B., Shen, H., Israr, T., Merseguer, J.: Performance by unified model analysis (PUMA). In: WOSP '05, ACM (2005) 1–12
9. Petriu, D.B., Woodside, M.: An intermediate metamodel with scenarios and resources for generating performance models from uml designs. In: Software and Systems Modeling, Volume 6, Issue - 2. (2007) 163–184
10. Knöpfel, A., Gröne, B., Tabeling, P.: Fundamental Modeling Concepts: Effective Communication of IT Systems. John Wiley & Sons (2006)
11. XJ Technologies: AnyLogic — multi-paradigm simulation software (2008) URL http://www.xjtek.com/anylogic/.
12. ATLAS Group: ATLAS transformation language (June 2007) URL http://www.eclipse.org/m2m/atl/.
13. OMG: MOF QVT Final Adopted Specification. (June 2005)
14. SmartQVT: SmartQVT - A QVT implementation (2008) http://smartqvt.elibel.tm.fr/.
15. Kolovos, D.S., Paige, R.F., Polack, F.: The Epsilon Object Language (EOL). In: ECMDA-FA. Volume 4066 of LNCS., Springer (2006) 128–142
16. Jouault, F.: Loosely Coupled Traceability for ATL. In: ECMDA-FA workshop on traceability. (2005)
17. Fabro, M.D.D., Bezivin, J., Jouault, F., Breton, E., Gueltas, G.: AMW: a generic model weaver
18. OMG: UML profile for schedulability, performance, and time specification (January 2005) URL http://www.omg.org/docs/formal/03-09-01.pdf.
19. Grassi, V., Mirandola, R.: A Model-driven Approach to Predictive Non Functional Analysis of Component-based Systems. In: NfC'04 Technical Report TR TUD-FI04-12. (2004)
20. Sabetta, A., Petriu, D.C., Grassi, V., Mirandola, R.: Abstraction-raising Transformation for Generating Analysis Models. In: MoDELS'2005 Satellite Events: Revised Selected Papers, LNCS 3844. 217–226