Towards Supporting Multiple Execution Environments for UML/OCL Models at Runtime (Position Paper)

Lars Hamann
University of Bremen
Bremen, Germany
lhamann@tzi.de

Martin Gogolla
University of Bremen
Bremen, Germany
gogolla@tzi.de

Daniel Honsel
University of Bremen
Bremen, Germany
dhonsel@tzi.de

ABSTRACT

Our approach allows a developer to verify whether a model corresponds to a concrete implementation in terms of the JVM (Java Virtual Machine) by validating assumptions about model structure and behavior. In previous work, we focused on (a) the validation of static model properties by monitoring invariants, (b) basic dynamic properties by specifying pre- and postconditions of an operation and (c) employment of protocol state machines for validating advanced dynamic properties.

This paper discusses the generalization of the underlying architecture for the JVM to easily incorporate other runtime environments like the CLR (Common Language Runtime). This is realized by extracting common features like method calls and identifying relevant interception points.

Keywords

UML model, OCL constraint, Validation, Runtime monitor, Virtual machine, JVM, CLR

1. INTRODUCTION

Lifting models to the runtime level has been identified as a promising way to handle complex systems at runtime, c. f. [2] and previous editions of the Model@Runtime workshop. While formal models are used to apply different verification and validation techniques, the inherent loss of semantics [1] when translating models to executable code adds some degree of uncertainty to the results of these tasks. By connecting models and their derived runtime manifestations the semantics present in the models can be validated. Our approach on runtime monitoring applications on the model level [9, 10, 11] uses UML [14] based models consisting of class diagrams and protocol state machines, enriched by constraints specified as OCL [15] expressions. Using our approach, the system under monitor (SUM) was previously forced to run on the Java Virtual Machine (JVM). In this paper we present ongoing work on how to overcome this limitation by adding an abstraction layer for different runtime targets.

The rest of the paper is structured as follows. Section 2 explains our monitoring approach from a bird's-eye view. In Section 3 the introduced abstraction to the runtime targets is explained. Before we conclude and take a look on future work in Section 5, we discuss related work in Section 4.

MRT 2012 Innsbruck, Austria

2. THE USE MONITORING APPROACH

Based on the stable UML/OCL tool USE (UML-based Specification Environment) [7] which is used as a validation tool by various audiences, i. a. [3, 4, 20, 21], we added runtime monitoring capabilities realized as a plugin [9, 11]. The first version of this monitor plugin realized a debugging approach [1] which takes advantage of the remote debugging capabilities provided by the Java Virtual Machine (JVM) to understand and to listen to the running application.

This allows a modeler to use all validation capabilities provided by USE to validate the runtime behavior of a system. She can validate

- the well-formedness of the specified UML/OCL model,
- structural mismatches, e. g., multiplicity violations,
- static constraints, i. e., invariants
- dynamic constraints defined as pre- and postconditions and
- protocol specifications for classes defined by protocol state machines [10].

The extracted snapshot can be examined using several visual and textual features provided by USE. To visualize the current snapshot, object diagrams can be used. To query a snapshot, OCL-expressions can be evaluated. Such queries can also be applied on object diagrams in order to select proper elements and to show or to hide them. Further, operation call sequences can be visualized by means of a sequence diagram.

Figure 1 shows a running monitor session in USE. Before we made the screenshot, we connected to the application (as indicated in lower left Monitor Control window), took a snapshot of the system state, resumed it and modified it by using the interface of the application. A small part of monitored system state can be seen in the middle of the figure as an object diagram showing relevant parts for the current application state. These relevant parts were extracted by using OCL queries to determine the shown objects in the object

diagram [8]. Placed at the lower center the current protocol state machine instance of a monitored object is shown. To the left of it you see the monitor control which allows to attach to an application, to pause and resume it and to stop the overall monitoring. At the right top of the screenshot the monitored operation call sequence after connecting to the application is shown.

3. RUNTIME ABSTRACTION

The previously described runtime monitoring approach was applicable only to applications running inside the Java virtual machine (JVM). As a next step we are currently integrating an abstraction layer to the monitor component to be able to monitor applications running on other target platforms then the JVM. The primary objective of this abstraction layer is to allow an easy adaption of our monitoring approach to different target platforms. One example for such a target platform is the Common Language Runtime (CLR) of Microsoft's .NET framework which we are currently looking at. This platform is of special interest for our approach, because several different technologies need to be connected. Our monitor and validation tool is running on the Java platform whereas the monitored system runs on the CLR. Further, the CLR provides COM interfaces to access its internals (we give details on this in section 3.1.3). While both platforms share common concepts, e. g., a stack-based object-oriented virtual machine, there are some features which are unique to each platform. One example is the possibility to define new value types in the CLR, whereas the JVM only supports the built-in value types (int, float, etc.).

To ease the task of adding a new target environment a clear separation of concerns for the involved components is needed. Before the runtime abstraction layer was introduced, only two components needed to be considered: the validation engine (VE) and the monitor. The validation engine was and is still independent of the monitor component. This allows for example the continues development of the VE without modifying the monitor. However, the users of the monitor can automatically benefit from new concepts introduced into the VE as it was done lately with the support for protocol state machines [10].

A major drawback of this architecture was the tight coupling of the monitor to a concrete runtime environment. Since major parts of the monitor are independent of a concrete runtime environment, we decided to add a new abstraction layer which is responsibly for the concrete communication with a specific target platform. Further, this abstraction allows other monitoring approaches [5], like for example, aspect oriented listeners to be supported.

To enable the monitor to communicate with any runtime, common data structures are needed. For this, we defined a small meta-model for virtual machines as shown on top in Fig. 2. The bottom part of the figure shows the elements from the USE meta-model and their relationship to the virtual machine elements. Not shown are the relations between them, but the USE elements nearly correspondent to the elements defined in the UML. Only the class[1] `MObject` has no direct representation in the UML specification, because its position in the standard four level meta-model architecture (M0-M3) of the OMG is on the runtime level. The separation of `MAttribute` and `MAssociationEnd` is an implementation related distinction. In UML 2 both elements are represented by the meta-class `Property`.

The classes placed on the top part of the VM meta-model area are used to map information from inside a virtual machine to USE. The three classes `VMType`, `VMMethod`, and `VMField` represent the structural part of a virtual machine, whereas the three classes `VMObject`, `VMMethodCall`, and `VMFieldModification` map the dynamic part. All classes can store an arbitrary identifier provided by the runtime environment, e. g., a unique number, to be able to determine the related element if new information is required. For example, it can be used to get the `VMObject` for a given field value.

3.1 Components and their Responsibilities

In the following subsections we describe the different components we use in our monitoring approach. Starting with the validation engine we show for each component their responsibilities.

3.1.1 Validation Engine

As already mentioned, we did not build a validation engine from scratch to be able to validate the runtime behavior of a system. Instead we based our approach on a long existing and stable validation engine called USE [7]. Due to its plugin framework it can be extended with new functionalities without the need to change the core implementation. Instead, the monitor plugin described next just adds another way to create instances of a given model. This architecture allows for an independent development of the VE. However, the development of plugins sometimes leads to new requirements for the plugin architecture, e. g., by adding new extension points or notifications.

3.1.2 Monitor

The monitor plugin is the heart of our monitoring approach. At first, it uses a pull mechanism, c. f. [22] to read a snapshot of the SUM. This is done by querying the configured `VMAdapter`. At the beginning, the adapter is used to get information about the structure of the system. For example, the inheritance tree of monitored classes is mapped to the inheritance hierarchy defined in the model. The monitor examines both inheritance models and, if required, maps multiple classes in the running system to a single class in the model. Instead of just ignoring these classes, this "inheritance tree compression" is needed because the instances of the subclasses might be used by other classes referring to the base class. We call this concept *abstracted superclass*, because it can be used to abstract a complex inheritance tree present in an implementation to a more simpler one. This is especially useful, if the concrete subclasses are of no interest in a given context. In the class diagram shown in Fig. 1 the class `Settlement` acts as an abstracted superclass to reduce the number of subclasses for this class. The implementation provides other specializations of Settlement, e. g., `IndianSettlement`, which are irrelevant for the validated aspects in this session. After this structural step, the monitor informs the adapter about notification interests for several changes. On the static level, the monitor might be interested if a new class definition is loaded, since most platforms load classes only if required. Notifications for runtime events are for example the creation of new instances or the modification of a field value. The last task for the monitor,

[1]We always use class here, although elements shown with a circle next to the name are interfaces.

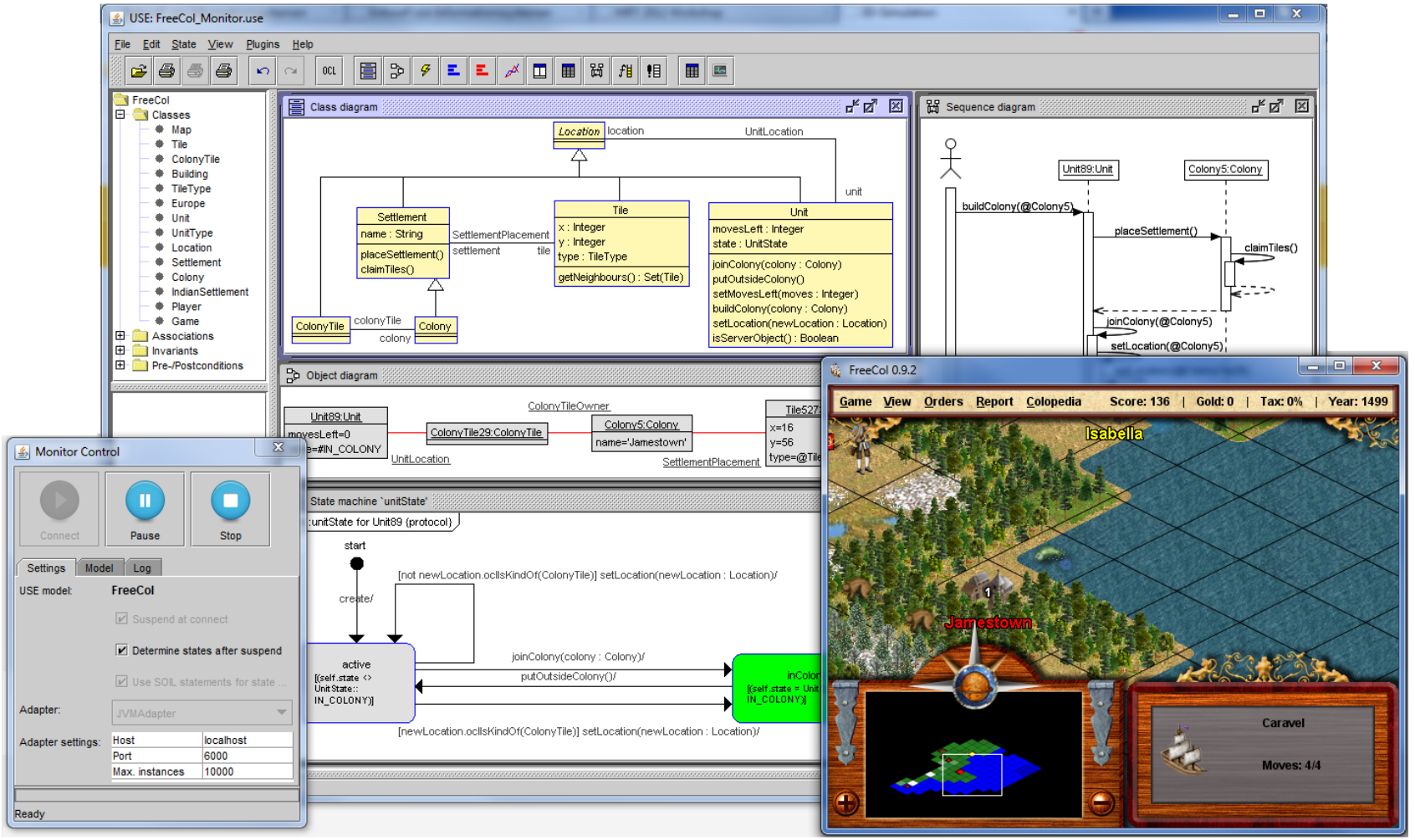

Figure 1: A USE model@runtime

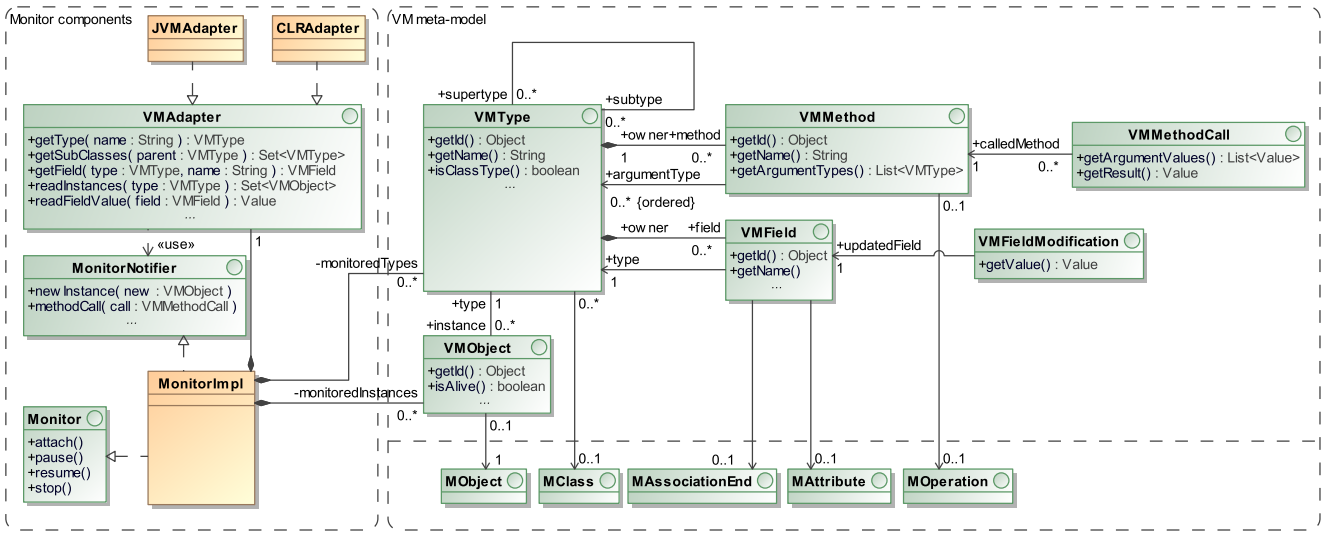

Figure 2: A Meta-Model for Virtual Machines

while operating in pull mode, is to extract a snapshot of the running system. For this, all instances of the runtime types previously identified as relevant are read and corresponding objects are created. Next, the field values of the runtime instances are either mapped to attributes or links in the runtime model. The monitor also keeps track of the runtime proxies [12] which can be identified by an id provided by the adapter. This id can be any Java object, which allows the adapter to store unrestricted information. For example, the JVM adapter stores objects returned by the Java debugger interface. This storage is needed to ease the identification task for instances, types, etc. needed while monitoring the dynamic behavior of the SUM.

After a snapshot has been taken, the monitor switches to passive mode waiting for push notifications [22] of the adapter, e. g., operation calls or field modifications to incrementally synchronize the snapshot with the running system. If the monitor receives such a notification it translates the received event to a command for the validation engine. The engine reacts on this call. For example, it validates valid transitions when receiving an operation call by examine the current state machine instance of the receiving object.

As can be seen in Fig. 2, the monitor class `MonitorImpl` implements two different interfaces: `Monitor` and `MonitorNotifier`. The former provides operations to control the overall monitoring process, while the latter provides operations required by the adapter. This separation is done to avoid that an adapter starts to control the monitoring process.

3.1.3 Adapter

An adapter is responsible for mapping platform specific entities to the meta-model shown in Fig. 2. To be able to configure unknown types of adapters, concrete implementations provide information about their settings. This can be seen in Fig. 3. The JVM adapter needs a hostname and a port to connect to an application, wheras the CLR adapter uses as process ID of a running process.

An adapter itself can be implemented stateless, because the `MonitorNotifier` interface provides operations to store and retrieve instances of the proxy classes. All operations needed for the pull-mode are defined in the interface `VMAdapter`. Also the operations called by the monitor to register for certain events are defined there. However, an adapter is not and cannot be forced to notify the monitor. Such an adapter can only be used to examine a current state of an application, leaving out dynamic aspects.

Beside the responsibilities which were implicitly introduced by the description of the monitor, an adapter has another major task. It needs to translate primitive runtime values to values understood by the validation engine. Because USE is based on UML/OCL the values need to be mapped to OCL values. USE provides Java classes for all types of OCL values which can be used by the adapter. Most of the values in a virtual machine are easy to translate, because OCL provides a corresponding type, e. g., String -> String or Array -> Sequence. Unfortunately, there is no support for map-like structures in OCL. One can simulate a map using qualified associations, but an adapter should be as generic as possible by providing OCL values if possible. This allows a modeler to use a map-like collection type as an attribute. For this, we expect an adapter to return an OCL value of the type `Set(Tuple(key:OclAny, value:OclAny))` for a map collection type. Doing so, the monitor can use the value either for creating links for qualified associations or to set an attribute value.

Using the described abstraction, concrete adapters can be implemented with reasonable effort. For example, the adapter for the JVM only consists of 350 lines of code. Because the implementation of the JVM adapter is already explained in [9] we only give some insights of the implementation of the adapter for the CLR in the next sections.

To gather runtime information about applications running in the Common Language Runtime, the CLR debug-

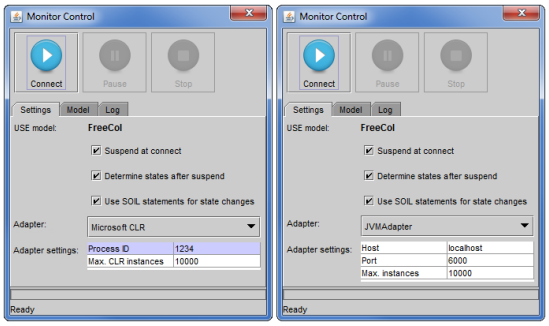

Figure 3: Adapter Configuration

ging API [13] is used. The debugging API consists of the Component Object Model (COM) interfaces, separated in objects that are implemented by the CLR and callback interfaces that must be implemented by the CLR adapter. Therefore, the main part of the adapter is written in C++. To integrate the C++ code and the Java adapter, the Java Native Interface (JNI) [16] is used. This is comfortable to define the interface between the native DLL and the Java CLR adapter. For this task, a Java class is extended with methods, that have to be marked with the keyword *native*. After compiling the Java code the tool *javah* is used to generate a C++ header file, that has to be implemented by the native part of the CLR-Adapter. This interface allows the instantiation and returning of Java classes, applying of Java objects as parameter, and Java exception handling from inside the native code. As a consequence, the native DLL is able to create all instances required by the monitor.

In order to attach the CLR adapter to a running system it requires the process ID (PID) of the application, that is monitored. This PID is used to open the selected process. After the process is opened, the main debug object, that implements the interface `ICorDebug`, will be initialized. This object is used to set the implemented callback handler and to get more specific debug objects. The callback handler provides runtime information, such as loaded assemblies, modules, classes, etc.(for more information about common CLR types cf. [17]). The debug objects support detailed information about the monitored application, such as heap information. Besides that, they allow the adapter to take control over the monitored application, such as: stop it, pause it and resume it. For the loaded modules some meta data information is available. Therefore, a pointer to each of the modules is stored in a set, which will be updated after a module is loaded or unloaded. Based on the meta data objects of the loaded modules, the CLR adapter takes a snapshot of the running application and maps instances inside the CLR to instances of the proxy classes.

Currently we are still working on improvements of the static features (snapshot of the heap). The performance is auspicious due to the usage of the shared memory. Further work will cover the dynamic part at runtime of the monitored application.

4. RELATED WORK

General work on connecting design time models to different runtime architectures was done in [12]. It separates *descriptive* and *prescriptive* parts for runtime models. Our meta-model for virtual machines does nearly conform to the proposed meta-metamodel of runtime-models presented there. However, we do not cover the prescriptive parts, yet.

The work in [22] proposes a model transformation engine to synchronize between a source model of a monitored system and multiple abstracted target models. The transformation is done by using TGG rules. The proposed source models need to be defined for each managed system. Our proposed meta-model for virtual machines can be seen as such a source model supporting multiple platforms.

The framework CALICO [23] also aims to be able to support multiple target platforms by using so called *platform drivers*. However, the tool focuses on component based development. The authors also propose an iterative design process which aligns well to our approach.

In [18, 19] the SM@RT tool is presented. Models based on ECore are synchronized by *synchronizers* which are generated by the tool. The SM@RT tool itself uses a pull mechanism only, i. e., when model elements are accessed, the synchronizer queries the runtime environment for the current state. Our approach offers also support for a push mechanism for adapters (synchronizers in the context of SM@RT), allowing an incremental built-up of the runtime model. However, SM@RT is capable of executing adaptations of the running system.

5. CONCLUSION

We have shown an extension to our previous work on runtime monitoring which allows an easier alignment of the monitoring process to different target platforms. For this, we concentrated on common features of object-oriented virtual machines. Our monitor plugin can easily be extended by providing new adapters which map elements of a runtime environment to the common features. The monitor uses this information to translate this information to UML elements. First attempts of creating an adapter for the Microsoft Common Language Runtime are promising. Therefore, we expect that our abstraction is applicable in general.

As future work we plan to compare different platforms and monitoring approaches by means of performance and usability. Further, adapters which indirectly map elements could be of interest. A possible use case for this is to monitor JRuby objects. For this, such an adapter needs to map over multiple language levels, because Ruby objects are represented as instances of a common Java class. A complete causal connection, i. e., reading and writing in both directions, in order to support model adaptations looks like a promising research direction. Especially, because USE already provides components encouraging the search for new configurations.

6. REFERENCES

[1] M. Balz, M. Striewe, and M. Goedicke. Monitoring Model Specifications in Program Code Patterns. In *Proc. of the 5th Int. WS Models@run.time*, pages 60–71, 2010.

[2] N. Bencomo, G. S. Blair, R. B. France, B. H. C. Cheng, and C. Jeanneret. Summary of the 6th international workshop on models@run.time. In J. Kienzle, editor, *MoDELS Workshops*, volume 7167 of *Lecture Notes in Computer Science*, pages 149–151. Springer, 2011.

[3] J. Brüning. Declarative Workflow Modeling with UML Class Diagrams and OCL. In W. Abramowicz, L. A. Maciaszek, R. Kowalczyk, and A. Speck, editors, *BPSC*, volume 147 of *LNI*, pages 227–228. GI, 2009.

[4] F. Büttner, M. Kuhlmann, M. Gogolla, J. Dietrich, F. Steimke, A. Pankratz, A. Stosiek, and A. Salomon. MDA Employed in a Joint eGovernment Strategy: An Experience Report. In T. Bailey, editor, *Proc. 3rd ECMDA Workshop "From Code Centric To Model Centric Software Engineering" (2008)*, `http://www.esi.es/modelplex/c2m/program.php`, 2008. European Software Institute.

[5] L. Froihofer, G. Glos, J. Osrael, and K. M. Goeschka. Overview and Evaluation of Constraint Validation Approaches in Java. In *Proc. of ICSE '07*, pages 313–322, Washington, DC, USA, 2007. IEEE Computer Society.

[6] S. Ghosh, editor. *Models in Software Engineering, Workshops and Symposia at MODELS 2009, Denver, CO, USA, October 4-9, 2009, Reports and Revised Selected Papers*, volume 6002 of *Lecture Notes in Computer Science*. Springer, 2010.

[7] M. Gogolla, F. Büttner, and M. Richters. USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming*, 69:27–34, 2007.

[8] M. Gogolla, L. Hamann, J. Xu, and J. Zhang. Exploring (Meta-)Model Snapshots by Combining Visual and Textual Techniques. In F. Gadducci and L. Mariani, editors, *Proc. Workshop Graph Transformation and Visual Modeling Techniques (GTVMT'2011)*. ECEASST, Electronic Communications, journal.ub.tu-berlin.de/eceasst/issue/view/53, 2011.

[9] L. Hamann, M. Gogolla, and M. Kuhlmann. OCL-Based Runtime Monitoring of JVM Hosted Applications. In J. Cabot, R. Clariso, M. Gogolla, and B. Wolff, editors, *Proc. Workshop OCL and Textual Modelling (OCL'2011)*. ECEASST, Electronic Communications, journal.ub.tu-berlin.de/eceasst/issue/view/56, 2011.

[10] L. Hamann, O. Hofrichter, and M. Gogolla. OCL-Based Runtime Monitoring of Applications with Protocol State Machines. In A. Vallecillo, J.-P. Tolvanen, E. Kindler, H. Störrle, and D. S. Kolovos, editors, *ECMFA*, volume 7349 of *Lecture Notes in Computer Science*, pages 384–399. Springer, 2012.

[11] L. Hamann, L. Vidács, M. Gogolla, and M. Kuhlmann. Abstract Runtime Monitoring with USE. In *Proc. CSMR 2012*, pages 549–552, 2012.

[12] G. Lehmann, M. Blumendorf, F. Trollmann, and S. Albayrak. Meta-modeling Runtime Models. In J. Dingel and A. Solberg, editors, *MoDELS Workshops*, volume 6627 of *Lecture Notes in Computer Science*, pages 209–223. Springer, 2010.

[13] Microsoft. Debugging (Unmanaged API Reference), 2012. http://msdn.microsoft.com/en-us/library/ms404520.

[14] *UML Superstructure 2.4.1*. Object Management Group (OMG), Aug. 2011.

[15] *Object Constraint Language 2.3.1*. Object Management Group (OMG), Jan. 2012.

[16] Oracle. Java^{TM}Native Interface, 2012. http://docs.oracle.com/javase/1.4.2/docs/guide/jni/.

[17] J. Richter. *CLR via C#*. Microsoft Press, Redmond and WA, 3 edition, 2010.

[18] H. Song, G. Huang, F. Chauvel, and Y. Sun. Applying MDE Tools at Runtime: Experiments upon Runtime Models. In N. Becomo, G. Blair, and F. Fleurey, editors, *Proceedings of the 5th International Workshop on Models at Run Time*, Oslo, Norvège, 2010. to be published.

[19] H. Song, Y. Xiong, F. Chauvel, G. Huang, Z. Hu, and H. Mei. Generating synchronization engines between running systems and their model-based views. In Ghosh [6], pages 140–154.

[20] W. Sun, E. Song, P. Grabow, and D. Simmonds. Xmi2use: A tool for transforming xmi to use specifications. In C. Heuser and G. Pernul, editors, *Advances in Conceptual Modeling - Challenging Perspectives*, volume 5833 of *Lecture Notes in Computer Science*, pages 147–156. Springer Berlin / Heidelberg, 2009.

[21] V. Thapa, E. Song, and H. Kim. An Approach to Verifying Security and Timing Properties in UML Models. In *Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on*, pages 193 –202, march 2010.

[22] T. Vogel, S. Neumann, S. Hildebrandt, H. Giese, and B. Becker. Incremental model synchronization for efficient run-time monitoring. In Ghosh [6], pages 124–139.

[23] G. Waignier, P. Sriplakich, A.-F. L. Meur, and L. Duchien. A Model-Based Framework for Statically and Dynamically Checking Component Interactions. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *MoDELS*, volume 5301 of *Lecture Notes in Computer Science*, pages 371–385. Springer, 2008.