Electronic Communications of the EASST
Volume 44 (2011)

Proceedings of the Workshop on OCL and Textual Modelling (OCL 2011)

OCL-based Runtime Monitoring of JVM hosted Applications

Lars Hamann, Martin Gogolla, Mirco Kuhlmann

20 pages

Guest Editors: Jordi Cabot, Robert Clariso, Martin Gogolla, Burkhart Wolff
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

OCL-based Runtime Monitoring of JVM hosted Applications

Lars Hamann[1], **Martin Gogolla**[2], **Mirco Kuhlmann**[3]

[1] lhamann@informatik.uni-bremen.de
[2] gogolla@informatik.uni-bremen.de
[3] mk@informatik.uni-bremen.de
University of Bremen, Computer Science Department
Database Systems Group, D-28334 Bremen, Germany

Abstract: In this paper we present an approach that enables users to monitor and verify the behavior of an application running on a virtual machine at the model level. Concrete implementations of object-oriented software usually contain a lot of technical classes. Thus, the central parts of an application, e.g., the business rules, may be hidden among peripheral functionality like user-interface classes or classes managing persistency. Our approach makes use of modern virtual machines and allows the devloper to profile an application in order to achieve an abstract monitoring and verification of central application components. We represent virtual machine bytecode in form of a so-called platform-aligned model (PAM) comprising OCL invariants and pre- and postconditions. In contrast to related work, our approach uses the original source or bytecode of the monitored application as it stands and does not require any changes. We show a prototype implementation as an extension of the UML and OCL tool USE. Also, we investigate the impact of our approach to the execution time of a monitored system.

Keywords: Runtime Validation, Monitoring, OCL, UML, Virtual Machine, Profile

1 Introduction

Model-driven development (MDD) is currently considered to be a promising paradigm for software production. MDD aims at employing models in all development phases and for different purposes. Quite common is the forward transformation of a platform-independent model (PIM) into a platform-specific model (PSM). Less common, but also studied is the backward direction transforming a PSM into a PIM. This paper studies the latter direction and concentrates on how to connect, monitor and analyse applications running on a virtual machine (e.g., the Java virtual machine (JVM) for Java or the common language runtime (CLR) for .NET languages) in terms of a design-like model formulated as a UML class diagram and enriched with OCL state invariants and OCL operation pre- and post-conditions [OMG09, OMG10].

The aim of our work is to detect general properties of a running application. When saying 'general', we think of properties that are not explicitly part of the source code but reflect characteristics which generalize and abstract certain implementation details. Our aim is to formulate central properties of a running application as OCL invariants and OCL pre- and postconditions. We call a collection of such properties a platform-aligned model (PAM) which can be seen as a link between a PSM and a PIM. A PAM will be formulated by means of assumptions which have

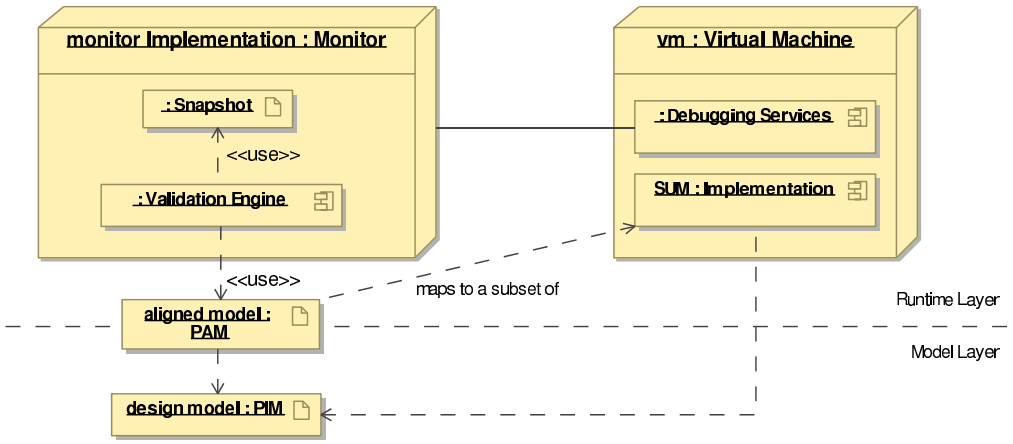


Figure 1: Deployment diagram of the monitoring approach

to be checked in prototypical scenarios invented and formulated by the developer. Designing a PAM is an iterative process in which assumptions are stated, checked and refined. Failure of an assumption may be due to an unjustified assumption which was made in the model or due to a justified assumption which does not hold in the implementation. According to the failure reason, one either has to change the model or report the failed assumption to the implementor. Thus, the development of a PAM may be seen as a (further) testing and quality assurance process for the running application.

The rest of this paper is structured as follows. In Section 2 we put forward the basic ideas of our proposal for analyzing applications running in the Java virtual machine. Section 3 explains these ideas by means of a middle-sized case study applied with a plugin for the tool USE [GBR07]. Section 4 examines the impact on the runtime performance of a system and shows details about special parts of our approach. Section 5 discusses related work. The paper ends with a conclusion and ideas for future work.

2 General approach

The main idea of our approach is to bridge the gap between platform independent models (PIM or abstract models) and the most platform specific models (PSM or implementation models). The bytecode of applications running inside a virtual machine can be seen as a PSM which is abstract enough to apply our approach, but also specific enough to make assumptions about the running system. This level of abstraction is needed because at this level one can make use of already existing features of the runtime environment of the PSM.

Modern virtual machine implementations like the JVM or the CLR of Microsoft .NET provide a rich pool of debugging and profiling interfaces. For example, the Java Platform Debugger Architecture [Ora11] allows easy access to applications running inside a (possible remote) virtual machine. We applied our approach to the Java virtual machine, but it should be possible to apply it to other virtual machines as well.

The first step of our approach is to define an platform aligned model (PAM) of the system under monitoring (SUM) which describes the expected behavior in a declarative way. This PAM could, for example, be generated out of a PIM, or reverse engineered out of an implementation. Further a PAM could be derived from a component specification to validate the possible externalized implementation of the component during the integration test phase. For this scenario our approach fits well because it does not need full access to the sourcecode of a component or system.

The PAM lies in between the runtime layer of an application and the modeling layer when using a model driven development process. Figure 1 shows the position and relations of the platform aligned model in the overall monitoring approach.

The PAM is provided as a UML model containing central classes of the SUM with attributes and associations. The class definitions contain relevant attributes, operations and OCL invariants. The dynamic behavior of a class is specified by means of OCL pre- and postconditions of the operations. The PAM should only contain central aspects of the SUM, i. e., it should abstract as far as possible from technical implementation aspects. To be able to monitor systems without modifying their source- or bytecode, the model needs to be enriched with annotations containing some information about implementation details. These implementation details are for example the concrete package a class is located in or a different name of an attribute. Further, query operations used inside the monitor need to be explicitly annotated because the monitor should not trace their execution inside the SUM.

The next step is to execute the SUM with enabled remote debugging capabilities. In the case of the JVM this can be done by providing specific arguments at startup. We do not make any assumptions about how the SUM is executed. Two possibilities are to execute it manually or by a test driver.

Once the SUM is started, the monitor with the PAM specified in the first step needs to be attached to the running system to start the monitoring process. In USE this is done by invoking a `monitor start` command with information how to connect to the remote application. The required information consists of the name of the host on which the application is running and the port on which the virtual machine is listening for a remote debugger. This port can be set as a startup parameter of the virtual machine. After the monitor has successfully connected to the SUM, it is left to the concrete implementation of the monitor, if the SUM is further executed or immediately suspended. However, the dynamic monitoring of a running SUM can only be done after it has once been suspended and an initial abstract snapshot of the system state has been taken. Such an abstract snapshot, e. g., an instantiation of a PAM, can be build up following these steps:

1. For all classes in the PAM which can be matched directly (by name or by special annotation information) to an already loaded class in the JVM[1], all existing instances in the JVM are mapped to newly created instances of the platform aligned model. In detail, this can be done by invoking the operation `instances()` on an object of the type `ReferenceType` which returns proxies to all reachable objects inside the JVM. This – for our approach important – operation was introduced in JVM version 1.6.

[1] Using the default class loader Java uses lazy initialization for classes. Therefore, not all classes might be loaded when building a snapshot.

2. For each created abstract instance in step 1 the attribute values are read. The mapping of primitive Java types to primitive OCL types should follow the common practice (c. f. [WK03]). Attributes with a type of a class defined in the PAM, i. e., reference types, can be read by using the mapping created in step 1. The possibility to define attributes referencing other instances is the reason why the creation of instances (step 1) and this step needs to be separated.

3. For all associations in the abstract model, links are created between corresponding instances. Technically this step can be merged into step 2 for performance reasons. The retrieval of links is discussed in Sec. 4.2.

After such a snapshot has been build, the monitor needs to register to several events that occur in the VM in order to allow a dynamic monitoring of the SUM. For example, the monitor needs to get informed if a not yet loaded class is initialized to be able to react on operation calls on instances of that class. However a user can already examine the SUM at this time by performing a check of the system state, e. g., by checking multiplicity constraints and invariants, by querying the system state with OCL expressions, or by visualizing the system state using examination patterns as described in [GHXZ11].

The next step in the monitoring process is to resume the suspended SUM to monitor its runtime behavior. In USE, this is done by simply invoking the command `monitor resume`. Now, a monitor can make use of the before mentioned events that it registers for. To keep the snapshot synchronized with the SUM, a monitor needs to set and listen to breakpoints inside the VM at several locations:

1. At class initialization to allow the registration of the breakpoints described next.

2. At constructors of monitored classes, i. e., classes defined in the abstract model. This allows the monitor to keep track of newly created instances and therefore enables an incremental built-up of the system state in contrast to always building a new snapshot of the running system when needed. Additional issues need to be considered for this dynamic build-up of the system state which are discussed later.

3. At the start of an operation which is specified in the abstract model. This enables the monitor to validate preconditions at runtime and in case of a failure pause the SUM.

4. Just before the exit of an operation call. This enables the monitor to validate postconditions. The break must occur after the result of the operation is calculated. The JVM provides such a mechanism. To reduce the total number of breakpoints the operation exit breakpoint can be set while entering a monitored operation and can be removed after the postconditions have been validated.

5. When a monitored attribute or link is modified. An application does not need to always use operations to modify attributes of an object. Therefore, a monitor needs the possibility to react on a modification of an object field to synchronize its snapshot. The JVM provides notifications when a field is modified to keep track of changing attributes or single values association ends. The monitoring of changes to many to many associations is more complicated and is discussed in Sec. 4.2.

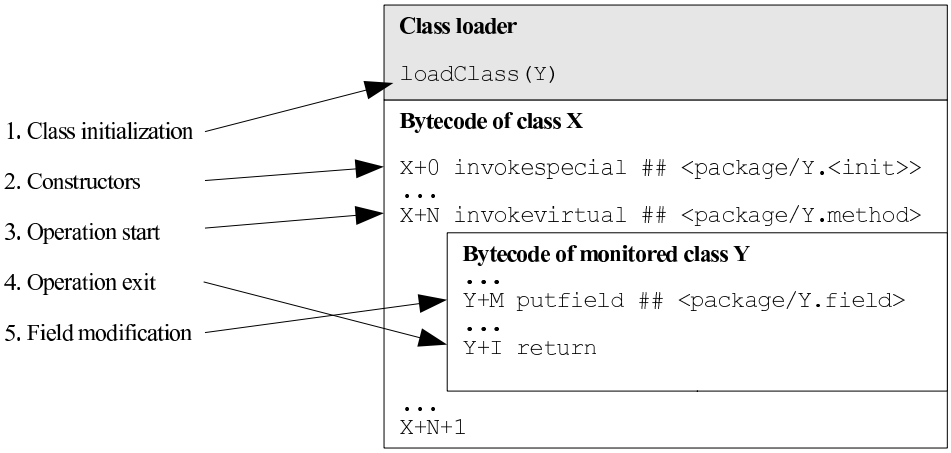


Figure 2: Monitoring events and the corresponding locations on the bytecode level

Figure 2 maps these listening locations to their adequate representation in Java bytecode, except the event when new classes are initialized. This event has no direct representation as a bytecode instruction and is also very specific to the virtual machine and the used class loader. Therefore, it is shown in an informative way.

These event locations allow a monitor to capture the relevant modifications inside a running application and trace its execution. This incremental build-up can be done until the application is exited or the monitoring process is ended. However, while applying this approach we found it useful to rebuild the snapshot when pausing the monitored application again. This enables the monitor to clean-up internal states.

Monitoring an application in the presented way allows a user to monitor the validity of UML constraints like multiplicities or compositions, invaraints, pre- and postconditions without the need to modify the source code of the application or to use special bytecode intersection mechanism which might alter the behavior of the system. A user can validate formulated assumptions about the application at runtime. This can be useful when validating a third party component where the sourcecode itself is not available, but the specification of the public interfaces can be used to create a PAM. When encountering an error during the monitoring process a user can make use of the, in contrast to the usage of a debugger, more abstract snapshot of the system. This more abstract snapshots focuses on the central parts of an application by hiding technical details. This task can be seen as *abstract debugging*. After locating the error, the user has to decide if the implementation or the PAM has to be corrected. This is equal to the task when testing and finding an error. To reduce the errors in the PAM, unit tests can be used as introduced for OCL in [CO09] and discussed in detail in [HG10].

3 Case Study

In this section we apply our monitoring approach to an existing mid-sized application using an developed plugin for the USE tool. We monitor the application to validate assumptions about

its structure and behavior. These assumptions are formulated by multiplicities, OCL invariants and OCL pre- and postconditions. Further, we show how the examination of a snapshot helps to explore unexpected behavior of a system, e. g., memory leaks.

We exemplify our approach by using an open source computer game called *Free Colonization*[2] or in short *FreeCol*. It is a modern Java-based implementation of the 1994 published game *Sid Meier's Colonization*[3]. The game itself is a round-based strategy game with the goal to colonize America and finally to achieve independence. The game takes place on a matrix-like map which consists of tiles with different types, e. g., water, mountain, forest. Different units operate on this map and can explore unknown territory, build colonies, trade goods, etc. Fig. 3 shows an example state of a running game. One unit (i. e. a pioneer) is placed in the center of the shown map part surrounded by several different tile types.

To formulate assumptions about the application we start by taking a look at some central game rules. While there are many other rules, we only use some rules related to the founding of a colony to keep the example moderate. The following rules are derived by examining the documentation and by own observations while executing the game. A unit can build a colony if

1. its current position is on a tile which does not contain another colony,
2. the unit has enough moves left to build a colony, or
3. there are no other colonies placed directly to the current tile.

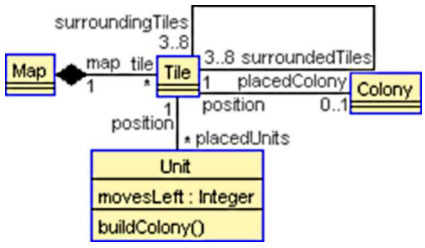
Because we are monitoring an existing application which does not provide a design model we need to build one from scratch. Another approach would be to reverse engineer the source-code and then simplify the extracted model to the required elements. As we will see, building a model from scratch does fit well to our purpose. When analyzing the rules using the common approach to find candidate classes by nouns, we find four class candidates in the rules: `Position`, `Tile`, `Colony`, `Unit`. However there are some other needed classes, e. g., `Map` which is not mentioned in the rules but the class is needed as a container. Other candidates are no classes but roles of them, e. g., position as role of tile.

A possible platform independent model which can be created out of the information given by the above rules is shown in Fig. 4(a). In this model a unit is positioned on a tile which is part of exactly one map. A tile has three to eight surrounding tiles and can be the position of at most one colony. The available moves of a unit a stored inside of the attribute `movesLeft`. Our assumptions about when a unit is allowed to build a colony are shown as OCL preconditions in Fig. 4(b).

As described before, the PIM has to be aligned to the platform the application is running on. Therefore information about the concrete implementation is needed. When applying our approach as part of a model driven process these information is encoded inside the transformation rules used to generate the PSM and can be reused to generate the PAM. While we are examine an application which is not developed in a model driven way, we need to align it manually by examining the implementation.

[2] Project website: http://www.freecol.org
[3] The corresponding Wikipedia article gives detailed information about the game play. http://en.wikipedia.org/wiki/Sid_Meier%27s_Colonization

Figure 3: Sample game situation in FreeCol



(a) Class diagram

```
context Unit::buildColony()
  pre tileIsEmpty:
    self.position.placedColony.isUndefined()

  pre noSurroundingColonies:
    self.position.surroundingTiles->forAll(t |
      t.placedColony.isUndefined())

  pre hasMovesLeft:
    self.movesLeft > 0
```

(b) Preconditions

Figure 4: Platform independent model derived from above game rules

The source code of version 0.9.2 of FreeCol contains an overall of 551 classes, but as we will show relevant to our goal to validate the implementation of the above rules are only few of them. The central "business logic" of FreeCol is located in a package called `net.sf.freecol.common.model`. This package still contains 92 classes. The concrete implementation differs from our first model because of various reasons. First, it takes into account a lot of other features which are not relevant to our assumptions. Further, the developers took other design decisions when implementing the game. For example the implementation of the map stores the tiles inside of a multi-dimensional array whereas we modeled it as some kind of linked list, i. e., the map is constructed by linking a tile to its surrounding tiles. From the modeling perspective, that makes sense, but taking performance considerations into account the array implementation fits better.

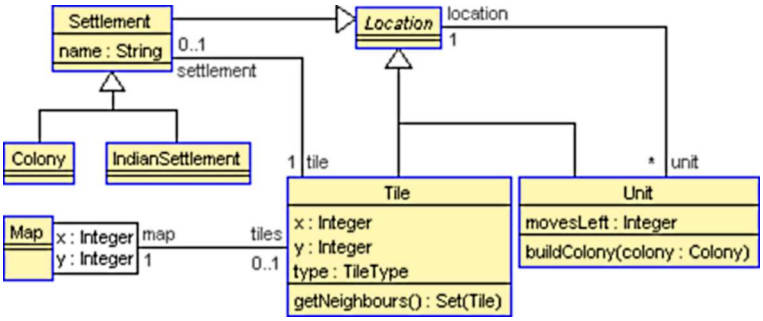
A model which is aligned to the concrete implementation is given in Fig. 5(a). One can see that the reflexive association of tile is no longer needed because the neighbored tiles can be calculated by the x and y coordinates. The implementation as a multidimensional array is represented as a qualified association which also guides the snapshot generation process to read an array at runtime. Another interesting change is the introduction of the class `Location`. While examining the rules we stated that position is a role instead of a class. It turns out that due other features a class `Location` is needed because there are several entities that can serve as a location. A unit itself can be the location of other units, e. g., a ship. Another important change is the introduced parameter `colony` of the operation `Unit::buildColony()`. The developers decided that not the class `Unit` should take care of creating a new instance of the class `Colony`. Instead, an already created instance is passed as an argument.

Because the structure of the model changed, the OCL constrains defined for the PIM need to be changed, too. The adjusted constraints are shown in Fig. 5(b). One might wonder why the invariant `Colony::noNeighbours` is contained in the model. Looking at the preconditions of the operation `buildColony()` it seems to be redundant. The reason for explicitly considering the invariant is that while monitoring, our approach allows a user to attach to a system at any time. Therefore we cannot make any assumptions about the validity of the preconditions in previous calls to operations.

The operation `Tile::getNeighbours()` is introduced to simplify the definitions of the constraints. To notify USE to ignore this operation while monitoring it is annotated as a query operation. This is done by the USE annotation mechanism that is provided to allow plugins to read additional information out of a USE model without the need to change the model parser. USE annotations look very like Java annotations. After an `@` symbol the name of the annotation is given following a possible empty list of attribute values pairs enclosed in brackets:

```
@Monitor(isQuery="true")
getNeighbours() : Set(Tile) = let neighbours = Set{} in ...
```

On the semantic level, these annotations are conceptually equal to UML stereotypes. The only difference in USE is that they are not statically typed, e. g., no profile has to be defined and referenced. The model can now be used to monitor the execution of the application. In contrast to simplify an automatically reversed engineered model with all 551 classes their attributes and operations which would have been reverse engineered, the demonstrated forward modeling approach resulting in seven classes seems to be more efficient when validating central aspects of a system.



(a) Class diagram

```
context Unit::buildColony(colony:Colony)
  pre movesLeft: self.movesLeft > 0

  pre tileIsEmptyAndFits:
    self.location.oclIsKindOf(Tile) and
    self.location.oclAsType(Tile).
      settlement.isUndefined()

  pre noSurroundingColonies:
    self.location.oclIsKindOf(Tile) and
    self.location.oclAsType(Tile).
      getNeighbours()->forAll(t |
        t.settlement.isUndefined())
```

```
context Colony inv noNeighbours:
  self.tile.getNeighbours()->forAll(t |
    t.settlement.isUndefined())
```

(b) Constraints

Figure 5: Platform aligned model

To begin the monitoring process the application needs to be started with additional parameters which setup the interfaces of the virtual machine to listen for remote connections. The parameters are well documented in the JVM documentation and are not described here, except one interesting parameter. The parameter `suspend` allows to specify the execution behavior of the virtual machine. When using the value `yes` the JVM immediately pauses execution until a remote application instruments it to resume. This option is useful to monitor an application including the whole initialization process.

After FreeCol is started with a JVM listening for a connection, the monitoring process can be started by USE. Before it can attach itself to the JVM the PAM has to be loaded. After this, the monitoring can be started by the command `monitor start`. After a successful connect, USE registers for important events and keeps track of changes inside the virtual machine. However when an application was started without the suspend option, USE at first needs a snapshot of the running application. This can be achieved by invoking the command `monitor pause`. USE suspends the monitored application and reads all instances of the classes specified in the PAM, sets their attributes and creates links as described in Sec. 2. Figure 6 shows parts of the snapshot taken at the state of FreeCol as shown in Fig. 3. We only show a part of it because already with the smallest map and at the very beginning of a game the snapshot read into USE consists of about 6,000 objects most of them (5,750) of type Tile and 4,000 links.

Please note that the alignment of the tile objects is following their x and y values and not their positions in the screenshot of the game. FreeCol uses a rather complicated approach following the layout on the screen to save the game maps. For example, when moving to north a unit decreases its x position by two instead of one.

While the colony *Isabella* and the Indian settlement can easily be found, the units are harder to identify because they are not named. `Unit85` is the Indian unit placed south of the Indian settlement. `Unit10` is the unit placed south-east of the Indian settlement. `Unit12` is the pioneer located in the center of the screen, whereas `Unit46` is not visible because it resides inside of the colony *Isabella* which is denoted inside of the screenshot by the number displayed in the center of the colony.

The difference between the number of tiles (5,750) and the overall number of links (4,000) already indicates that our assumption about the multiplicity specification at the association end `map` reachable from `Tile` is wrong. When examining the snapshot it turns out, that 1,830 tiles are not linked to a map but are referenced inside the virtual machine by some other objects. A possible cause of such a situation could be an implementation which leads to memory leaks. Although Java uses a garbage collector (GC) to reduce the possibilities of memory leaks, they still can happen. For example, when using static container classes the containing objects will never be collected by the GC because they are always reachable by the static container. In fact, the detection of memory leaks was one of the reasons why the used operation `instances()` was added to the JDA[4].

In our example, we used the following approach to examine the cause of the missing links to a map. In a step wise manner, we added classes to the PAM which use an attribute of the type `Tile`. For each step we connected to the a running game and took a snapshot of the running system and evaluated OCL queries on it. We quickly found classes which use delegates

[4] See http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=5024119

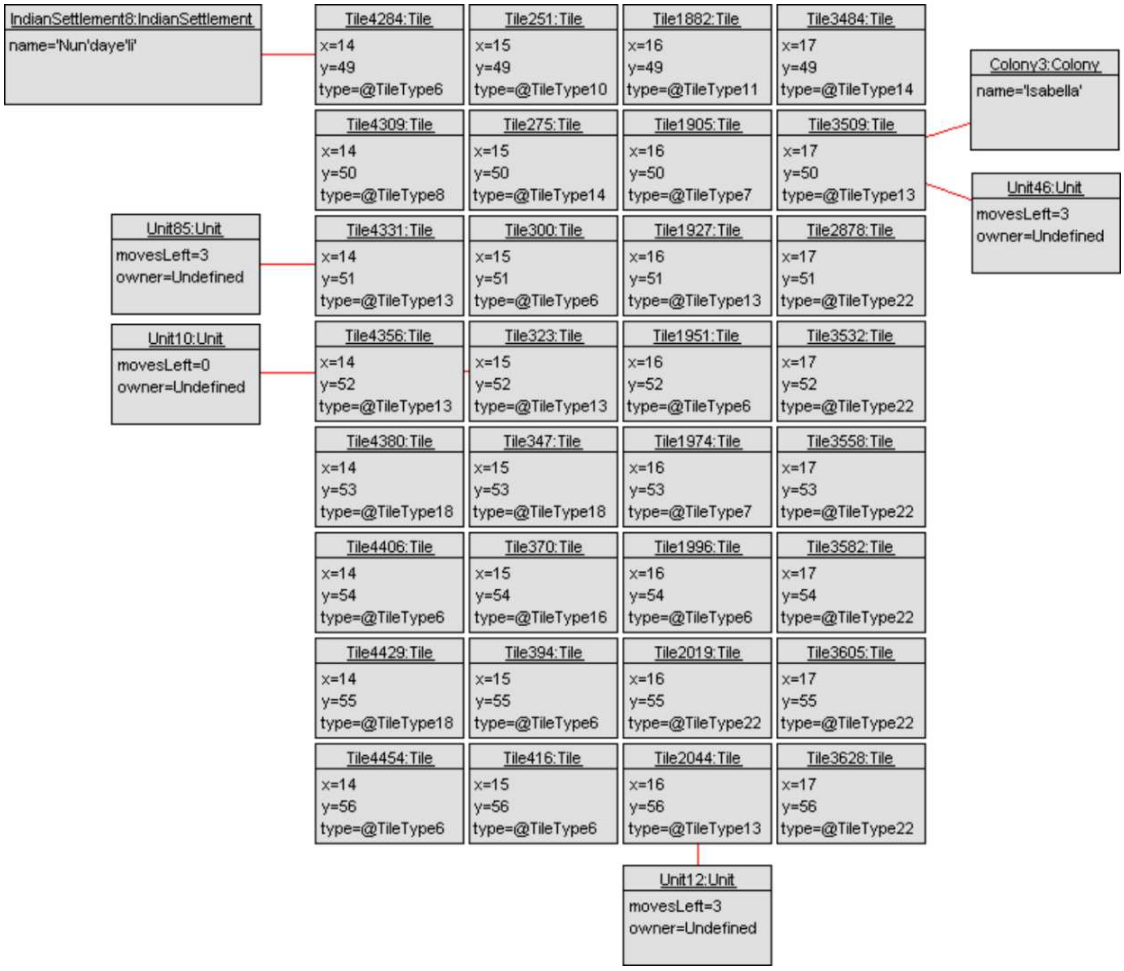


Figure 6: Parts of the snapshot taken at runtime

of tiles not connected to a map. These classes are mostly used inside of the graphical part of the application and are not used by the data model containing the important rules for our assumptions. Therefore, we could exclude a memory leak at this part and needed to align our assumption about the multiplicities. Interestingly, there seems to be still a memory leak related to the class Tile. After loading a saved game the number of tile instances is growing. We have not examined this issue any further, but it indicates that when loading a game the old game state is not disposed correctly.

While we have shown that examining a snapshot of a suspended application can be useful to detect possible structural issues, it can be used to examine some dynamic aspects of the application as well. One can check, for example, if an operation can currently be called on any instance of the defining class. Taking our snapshot into account one can check if any unit can currently build a colony. This can be achieved by using the preconditions as query conditions. However this is only possible in a simple way for preconditions that do not use parameter values. A skeleton for the combined query representing the precondition of the operation `Unit::buildColony` for all units owned by the player 'lhamann' is shown below. Instead of repeating the bodies of the preconditions shown in Fig. 5(b) they are represented by the placeholder `<preBody>`. The variables used inside of the let expressions denote the corresponding body.

```
let myUnits = Unit.allInstances()->select(owner.name='lhamann') in
  myUnits->select(self |
    let preMovesLeft = <preBody> in
    let preTileIsEmptyAndFits = <preBody> in
    let preNoSurroundingColonies = <preBody> in
      preMovesLeft and preTileIsEmptyAndFits and preNoSurroundingColonies)
```

This query results in a set of units which should be able to build a colony w.r.t. our assumptions. To validate our assumptions we resume the game and let the unit placed in the center of the sample state build a colony. Using our assumptions, this is indeed successful. The overall command list can be examined in USE and is shown in Fig. 7. Note that the object identifier are different to the identifier of the snapshot shown in Fig. 6, although the operation was called exactly at the same state. This is because we used a different run of the application to record the operation call using a saved game to start at the same state. This exemplifies, that when taking a snapshot one can not rely on the order in which instances are read, because the virtual machine could, for example, have reordered the objects on the heap.

The shown command list leads to another interesting observation. Some commands are executed more then once, e. g., setting the attribute `movesLeft` to 0. One can now examine the implementation to work out why this command is executed that often or she can refine the model to include more operation calls that should be monitored. When using the latter approach we quickly find out that several operations are setting the attribute value to zero. This behavior is indeed needed, because the operation can be called independent from each other.

Because the monitored product FreeCol is in a stable state of development and the observed operation is a central part of it, it is hard to identify a real bug to show a failing precondition. To simulate it, we interspersed a simple error (changing `movesLeft>0` to `movesLeft=0`) into our assumed precondition. Given this circumstances the last visible command in the command

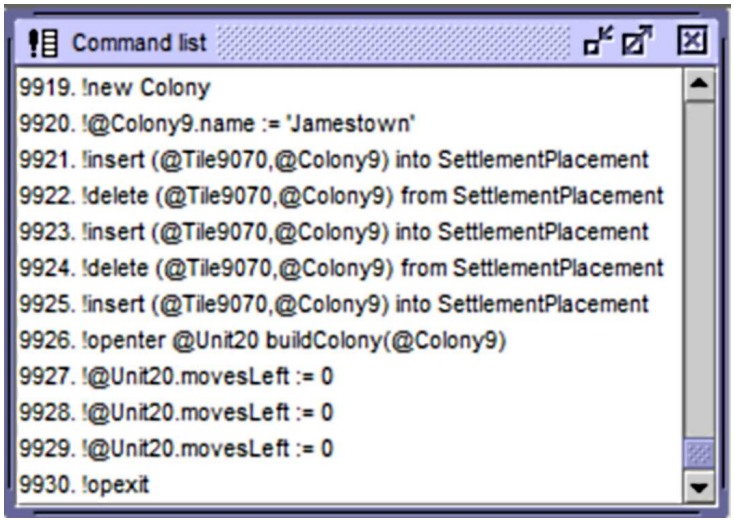


Figure 7: Monitored commands of buildColony()

list shown in Fig. 7 is 9925. A user now can examine the current system state and try to identify the error. As mentioned before the user has to take the specification of the PAM and the implementation into account and needs to judge what caused the error: a flawed implementation or incorrect assumptions as it is the case with our incorrectly defined precondition.

It could also be the case, that the design of an application uses a defensive programming style, i. e., the called operation validates it parameters and informs the caller of the failed preconditions by raising an exception. Therefore, in our approach the normal execution can be continued by resuming the application. Using such a defensive programming style will move the assumptions specified in a PAM into the postconditions, e. g., forcing the return value of an operation to the undefined value when an argument violates assumptions.

As with the preconditions, the handling of postconditions is nearly the same, except the access to the system state before the operation was called using the `@pre` operator. When using and OCL validation engine which supports the `@pre` operator and manages an own instance of the system state, this feature can be used without much effort. This is one reason, why the validation of constraints is done with an own snapshot instead of querying the Java heap.

When running the monitoring process with a more detailed PAM, the overall call stack can be taken into account when resolving failed assumptions. Call stacks can be visualized using a UML sequence diagram as shown in Fig. 8. Again, the object identifier changed because we needed to reattach to the SUM with a more detailed model. This visualization of call sequences is in our opinion also useful for documentation purposes. It allows an easy way to show central operation calls of real executions of a system, in contrast to exemplified call sequences constructed by hand or reversed engineered sequence diagrams showing an abstract execution path.

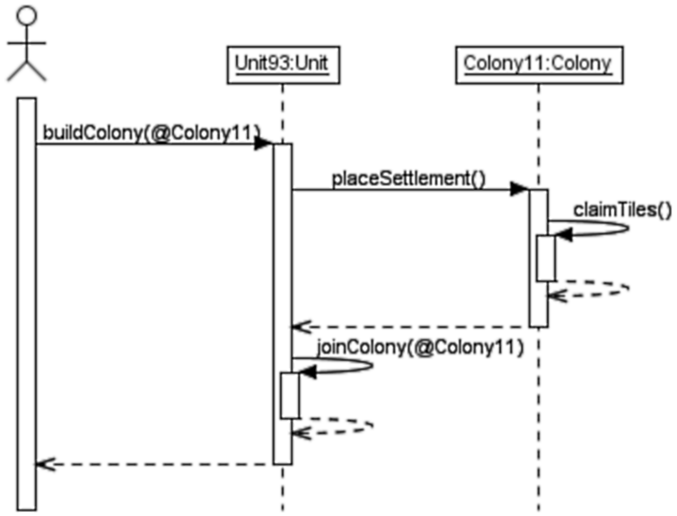


Figure 8: Monitored sequence diagram of an execution of buildColony()

Table 1: Performance of snapshot creation

Task	SOIL	Native	
Instance creation	≈8,700	≈9,700	instances/s
Attribute assignment	≈8,700	≈17,400	attributes/s
Link creation	≈4,100	≈4,100	links/s

4 Discussion

In this section we discuss technical aspects of out approach in detail. First give some brief information about general performance and the runtime overhead introduced by using our monitor implementation. After this, we discuss the link retrieval task in detail to show various ways with their advantages and disadvantages how to achieve this.

4.1 Performance and Runtime Overhead

Our implementation can use two different kinds of snapshot generation. It can be built by either using native USE system operations or by evaluating SOIL[5] statements [Büt11]. Using SOIL statements, the whole build-up process of the snapshot is encapsulated in command objects. These commands can be used to save an initial snapshot to a script file for later use. Table 1 shows the average values for the three main task when creating a snapshot, i. e., instance creation, attribute assignment and link insertion.

The values were measured on a Intel Core 2 Duo notebook running at 2.5 GHz while taking the whole snapshot which is partly shown in Fig. 6. The snapshots were taken several times to exclude the overhead of the just in time compiler. It can be seen that the impact of SOIL

[5] SOIL is an acronym for simple OCL-based imperative language.

Table 2: Performance of dynamic monitoring

Monitored events	Duration	#Events monitored	#Events/s
None (no monitor attached)	6 ms	0	n/a
None (monitor attached)	6 ms	0	n/a
Instance creation	≈7,600 ms	10,001	≈760
+ Attribute assignment	≈8,500 ms	30,002	≈3,530
+ Link creation	≈9,400 ms	40,002	≈4,225
+ Operation call	≈18,000 ms	60,002	≈3,333

comes to play only while assigning attribute values. This is due to the fact that an assignment of an attribute needs fewer validation tasks when executed than a link creation and therefore the encapsulation of the commands has a greater influence.

To examine the overhead of the dynamic monitoring we used a small application which executes several steps that can be monitored in a loop. We used an own small application because it allows a more precise measurement of the overhead in contrast to our case study which monitored an operation that is called rarely. For the case study we can only state that there is a marginal impact to the runtime behavior which leads to small delays that are barely noticeable, for example, when moving units, which changes parts of the snapshot, e. g., the unit position.

The application creates a new instance and calls an operation on it inside each iteration. The operation sets a primitive attribute of type integer and an object valued attribute. The loop was iterated 10,000 times. The time needed to execute the whole iteration with different granularity of monitored events is shown in Tab. 2. The overhead of one or two events respectively results from the fact that a single instance is created before the loop which is used to set the object valued attribute. When monitoring attribute assignments for each iteration step, two events are monitored: the initialization inside the constructor and the assignment inside the operation.

At a first look, this overhead seems to be out of scale, but as described before our approach is meant to be applied only to central parts of a system. Unrelated parts of the system are not tangled by the monitoring, e. g., graphical operations which are called very often, and therefore perform as without an attached monitor.

4.2 Link retrieval

While retrieving links of one-to-many associations can easily be done by reading the value of the field at the association end with multiplicity one, reading many-to-many associations is more complicated. This is similar to the issue how to generate association implementations when applying model transformations in an MDA process (c. f. [AHM07]).

We identified two potential ways to read links of a many-to-many association into a snapshot of an platform aligned model, either by examine the fields of the container object which saves the corresponding objects or by using iterators.

The main drawback of reading the details of container classes is that it requires a deep knowledge about the internal structure of them. Further, when new versions of the collection library, e. g., a new Java runtime version, is released the monitoring framework has to be adopted.

The usual way to abstract from these detailed information is to use some kind of iterator pattern [GHJV95]. However using a iterators requires to execute parts of the application out of the normal program flow. While this could be done with current virtual machines this could lead to forged results when monitoring an application. An implementation can for example write something while iterating over a container, but our monitoring approach should not alter the system state of a running application. The main benefit of this approach is that a monitored application does not need to keep all linked objects in memory at once, e. g., they can be stored in a database and retrieved when needed. We decided to retrieve links only by examining the fields of container classes to keep the execution flow of the monitored application untouched. However, we plan to support both approaches in the future.

Nearly the same considerations are valid in the context of the dynamic built-up of many-to-many links during program execution. A monitor can listen for a modification of the underlying data structure or it can set breakpoints at operations which modify the content of a container, e. g., `List.add(Object o)`. Which technique to use depends on the concrete implementation of the monitored system. For example, if the monitor uses operation breakpoints no detailed knowledge about the underlying container is needed, but it cannot be sure that an element is really added. This would be the case when using modification events, but as stated above a mapping to the concrete implementation is needed.

5 Related Work

Today, several approaches to applying runtime monitoring for verification and validation purposes exist. General comparisons regarding different methods for checking constraints at runtime have been carried out in [FGOG07] and [ASCY10]. The authors in [FGOG07] call approaches using AspectJ and other reactive techniques like proxy implementations 'Interceptor Mechanisms'. These mechanisms are related to our approach. However, all presented interceptor mechanisms alter the implementation of the monitored system, either by changing the sourcecode, by injecting bytecode, or by enforcing a particular architecture like the application of proxy classes.

In [ASCY10], the authors identify four distinctive approaches using OCL constraints to performing runtime checks:

(1) using implementation languages such as Java,
(2) using built-in assertion facilities such as the assert statement,
(3) using assertion or design-by-contract languages such as JML,
(4) using aspect-oriented programming language such as AspectJ

The first two categories are based on built-in structures of the target platform like `if`- or assertion statements. In contrast to our approach, the integration of approaches belonging to these categories into a system requires a full access to the sourcecode.

The Java Modeling Language (JML) can be applied for formal verification and runtime assertion checking [LCC+05]. Approaches for translating OCL expressions and constraints into JML are, for example, presented in [Ham04] and [AFC08]. In [CLSE05], program code is separated from code intended for specification purposes by introducing model methods and model fields

which abstract from concrete program variables and query methods. The respective features were implemented in the runtime assertion checker for JML. A JML compiler built on the Eclipse Java compiler is presented in [SC10] which, in contrast to the original JML compiler, supports Java 5 features, and is significantly faster, since it makes use of an AST merging technique.

The tool 'ocl2j' enforces OCL constraints in Java through translating OCL expressions into Java code [DBL06]. The generated assertion code is integrated at the bytecode level using AspectJ. Analog approaches are presented in [BDL05] (focusing on templates for automatically integrating invariants and pre- and posconditions at the bytecode level) and [GR08, GM09, RG03]. In [CA10], the AspectJ approach is applied to program testing by using OCL constraints for filtering test data and determining test results.

The Dresden OCL toolkit provides for two distinctive approaches to runtime verification based on OCL constraints [DW09]. Within the so-called interpretative approach the Dresden OCL2 Interpreter is integrated into a runtime environment interpreting the OCL constraints for all instances of the underlying model during execution. The 'generative' approach is currently based on the generation of AspectJ code which can ensure constraints at software runtime.

In [BSG10] the monitoring of state machines is focused. OCL is not used. The authors, though, sketch three general possibilities to extract runtime models. Beside the already mentioned 'aspect oriented approach', a so called 'listener approach' and a 'debugging approach' is described. The debugging approach is closely related to our method of using the debugging facilities. However, the tool presented in [BSG10] relies on the listener approach which can be seen as an architecture enforcing approach.

So called 'synchronizers' are used in [SHCS10] to synchronize a running system with a runtime model, i. e., to immediately change the system when the model has been updated, and to immediately adapt the model if the system progresses. Synchronizers can be generated for specific platforms. They make use of the APIs provided by the target systems. As discussed in Sec. 4.2, the use of APIs may lead to side-effects while querying the system state. In [SHCS10], the runtime model is represented in form of an EMF model. Thus, various MDE tools can be applied.

6 Conclusion

We presented an approach for monitoring assumed properties in form of OCL constraints for a running Java application. The approach was made possible by taking advantage of the powerful features of the Java virtual machine. Assumptions are formulated as state invariants or operation contracts and are understood as a platform-aligned model (PAM). We reported on a prototypical implementation of a monitor integrated into the UML-based Specification Environment (USE). The connection between the PAM and the platform-specific model (JVM byte code) was established through particular annotations in the PAM. Our approach does not need to modify the PSM as in approaches based on aspect-orientation. We explained our work by a non-trivial example of an open-source game.

As future work we want to (semi-)automatically detect the constraints in the platform-aligned model. For example, it could be possible to extract invariants or pre- and post-conditions (or at least parts thereof) from boolean expressions in the source code. The extraction of classes,

attributes and role ends of associations could be based on run-time metrics. We have to work further on the detection of associations and links in the case of many-to-many relationships. Comprehensive case studies will help to improve our work. An in-depth comparison to related approaches, for example, based on aspect-orientation or approaches considering the JML as a target language is needed. The prototype has to be improved in various directions. Moreover, a direct integration of OCL-like features into a virtual machine (e.g., by means of the plugin-like agent mechanism in the JVM) seems a promising line of research as well.

Bibliography

[AFC08] C. Avila, G. Flores, Y. Cheon. A Library-Based Approach to Translating OCL Constraints to JML Assertions for Runtime Checking. In Arabnia and Reza (eds.), *roceedings of the 2008 International Conference on Software Engineering Research & Practice, SERP 2008*. Pp. 403–408. CSREA Press, 2008.

[AHM07] D. Akehurst, G. Howells, K. McDonald-Maier. Implementing associations: UML 2.0 to Java 5. *Software and Systems Modeling* 6(1):3–35, mar 2007.

[ASCY10] C. Avila, A. Sarcar, Y. Cheon, C. Yeep. Runtime Constraint Checking Approaches for OCL, A Critical Comparison. In *roceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering (SEKE'2010)*. Pp. 393–398. Knowledge Systems Institute Graduate School, 2010.

[BDL05] L. C. Briand, W. J. Dzidek, Y. Labiche. Instrumenting Contracts with Aspect-Oriented Programming to Increase Observability and Support Debugging. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*. Pp. 687–690. IEEE Computer Society, Washington, DC, USA, 2005.

[BSG10] M. Balz, M. Striewe, M. Goedicke. Monitoring Model Specifications in Program Code Patterns. In *Proceedings of the 5th International Workshop Models@run.time*. Pp. 60–71. 2010.

[Büt11] F. Büttner. *Reusing OCL in the Definition of Imperative Languages*. PhD thesis, University of Bremen, 2011.

[CA10] Y. Cheon, C. Avila. Automating Java Program Testing Using OCL and AspectJ. In *Proceedings of the 2010 Seventh International Conference on Information Technology: New Generations*. ITNG '10, pp. 1020–1025. IEEE Computer Society, Washington, DC, USA, 2010.

[CO09] J. Chimiak-Opoka. OCLLib, OCLUnit, OCLDoc: Pragmatic Extensions for the Object Constraint Language. In Schürr and Selic (eds.), *Model Driven Engineering Languages and Systems*. Lecture Notes in Computer Science 5795, pp. 665–669. Springer Berlin / Heidelberg, 2009.

[CLSE05] Y. Cheon, G. Leavens, M. Sitaraman, S. Edwards. Model variables: Cleanly Supporting Abstraction in Design By Contract. *Softw. Pract. Exper.* 35:583–599, May 2005.

[DBL06] W. J. Dzidek, L. C. Briand, Y. Labiche. Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java. In *Satellite Events at the MoDELS 2005 Conference, MoDELS 2005*. LNCS 3844, pp. 10–19. Springer, Berlin, 2006.

[DW09] B. Demuth, C. Wilke. Model and object verification by using Dresden OCL. In *Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice*. Pp. 687–690. Ufa, Russia, 2009.

[FGOG07] L. Froihofer, G. Glos, J. Osrael, K. M. Goeschka. Overview and Evaluation of Constraint Validation Approaches in Java. In *Proceedings of the 29th international conference on Software Engineering*. ICSE '07, pp. 313–322. IEEE Computer Society, Washington, DC, USA, 2007.

[GBR07] M. Gogolla, F. Büttner, M. Richters. USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming* 69:27–34, 2007.

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[GHXZ11] M. Gogolla, L. Hamann, J. Xu, J. Zhang. Exploring (Meta-)Model Snapshots by Combining Visual and Textual Techniques. In *Proc. 10th Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'2011)*. 2011.

[GM09] S. R. GY. Cheon, C. Avila, C. Munoz. Checking design constraints at run-time using OCL and AspectJ. *International Journal of Software Engineering* 2(3):5–28, 2009.

[GR08] M. Gopinathan, S. K. Rajamani. Runtime Monitoring of Object Invariants with Guarantee. In *Runtime Verification, 8th International Workshop, RV 2008*. LNCS 5289, pp. 158–172. Springer, Berlin, 2008.

[Ham04] A. Hamie. Translating the Object Constraint Language into the Java Modelling Language. In *Proceedings of the 2004 ACM symposium on Applied computing*. SAC '04, pp. 1531–1535. ACM, New York, NY, USA, 2004.

[HG10] L. Hamann, M. Gogolla. Improving Model Quality by Validating Constraints with Model Unit Tests. In *Proc. 7th Int. Workshop on Model-Driven Engineering, Verification, and Validation (MODEVVA'2010)*. 2010.

[LCC+05] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program.* 55(1-3):185–208, 2005.

[OMG09] *UML Superstructure 2.2*. Object Management Group (OMG), Feb. 2009. http://www.omg.org/spec/UML/2.2/Superstructure/PDF/

[OMG10] *Object Constraint Language 2.2*. Object Management Group (OMG), Feb. 2010. http://www.omg.org/spec/OCL/2.2/

[Ora11] Oracle. Java™Platform Debugger Architecture - Structure Overview. 2011. http://download.oracle.com/javase/6/docs/technotes/guides/jpda/architecture.html

[RG03] M. Richters, M. Gogolla. Aspect-Oriented Monitoring of UML and OCL Constraints. In Aldawud et al. (eds.), *Proc. UML'2003 Workshop Aspect-Oriented Software Development with UML*. Illinois Institute of Technology, Department of Computer Science, http://www.cs.iit.edu/~oaldawud/AOM/index.htm, 2003.

[SC10] A. Sarcar, Y. Cheon. A new Eclipse-based JML compiler built using AST merging. Technical report 10-08, Department of Computer Science, The University of Texas at El Paso, Mar. 2010.

[SHCS10] H. Song, G. Huang, F. Chauvel, Y. Sun. Applying MDE Tools at Runtime: Experiments upon Runtime Models. In *Models@run.time*. Pp. 25–36. 2010.

[WK03] J. Warmer, A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 2003. 2nd Edition.