Abstract Runtime Monitoring with USE

Lars Hamann
University of Bremen
Germany
lhamann@informatik.uni-bremen.de

László Vidács
University of Szeged
Hungary
lac@inf.u-szeged.hu

Martin Gogolla
University of Bremen
Germany
gogolla@informatik.uni-bremen.de

Mirco Kuhlmann
University of Bremen
Germany
mk@informatik.uni-bremen.de

***Abstract*—We present a tool that permits developers to monitor and verify assumptions at an abstract level about an application running on a virtual machine. On the implementation level, a so-called platform aligned model (PAM) described in the UML (Unified Modeling Language) and enriched by OCL (Object Constraint Language) requirements is used to formalize these assumptions. Our solution allows a developer to concentrate on verifying core parts of an implementation while ignoring major parts of peripheral technical details. In order to easily detect a PAM which characterizes the central requirements, we propose a semi-automatic approach. First, a complete program model is generated by analyzing the source code. Afterwards, this model is reduced by the user to central classes and associations. This reduced model is enriched by the assumptions about the expected behavior of the system. The monitor connects to the running system at a particular point in time and builds up an abstract snapshot, i.e., an instance of the PAM, which corresponds to the current state. When the application is further executed this snapshot is synchronized by listening to changes in the running system. During monitoring the stated assumptions are validated and possible violations are reported to the user.**

I. Introduction

Formal models are playing an important role in several areas of software engineering. For example in model checking a formal model of a system has to be specified to verify properties of a system. This formal model needs to be a valid abstraction of the concrete system to be able to exclude errors which are introduced by the modeling task. In this paper we present a tool-chain which is able to simulate an abstract model by executing and monitoring its related implementation. The model is automatically built from the source code of an implementation. To be able to focus on central parts of the application we present filtering techniques to reduce the overall size of the system model. This reduced model is enriched by the user with system assumptions which are formulated as OCL [1] invariants, pre- and postconditions. The last model is called a platform aligned model (PAM). If the monitor detects a violation of a formulated assumption at runtime the user is informed. She can now explore the monitored system in an abstract way to identify the cause of the violation.

Several approaches on runtime verification exist. A detailed comparison of runtime verification approaches using OCL can be found in [2]. None of them uses the events provided by a virtual machine to react on changes in the monitored system. A runtime monitoring approach using other formal languages is for example JavaMOP [3].

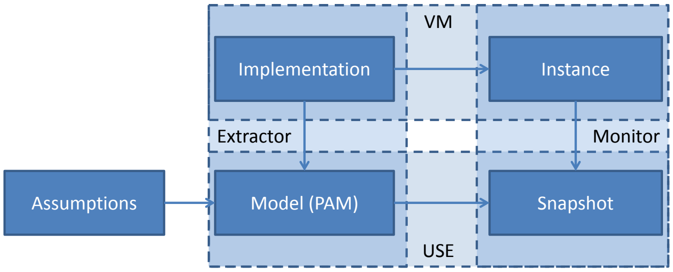


Fig. 1. Tools and artifacts

II. Monitoring Approach in USE

The tools and artifacts used in our approach are shown in Fig. 1. The artifacts are shown as solid rectangles. The dashed rectangles are the tools and cover the artifacts which are required by them.

A central part of our monitoring process is the UML and OCL tool USE [4] placed at the bottom of the figure. It uses a model and an instance of such a model to validate constraints included in the model. In the context of our monitoring approach the model instance is called a snapshot. In the upper part of the figure a virtual machine (VM) is shown, which executes an implementation. The runtime data of the application (heap space, stack frames, etc.) is labeled as the ‘instance’ of the implementation. The required parts of an instance are read by the monitor and transformed into a snapshot, i. e., an instance of the model provided to USE. This model is generated by examining the source code of the implementation and reducing it to central aspects of the system. The central aspects depend on the assumptions a user wants to check. Therefore, the generation task includes several steps. Some of them can be done automatically while others need user interaction. The result of this extraction process is a model which concentrates on properties of the system by ignoring irrelevant parts. After the extraction step the model is enriched with assumptions that should be checked at runtime.

A. USE (UML-Based Specification Environment)

The main task for USE originally was to support the design of systems in an early stage of development. A developer can specify a model of a system with a subset of UML and extend is with constraints formulated in OCL (Object Constraint Language, c. f., [1]). The formulated constraints can be validated by creating system states also called object diagrams and examining the evaluation result of the constraints against the specified system states. These system states can be built manually as scenarios to validate if the specified model behaves as expected. This manual checking is similar to unit tests on the source level. Further, formal verifications can be done to a certain degree by using a built-in system state generator [5]. The most simple verification is to check if an instance of a model exists, i. e., if the model is consistent. To verify this, a user can search within predefined bounds for a valid system state. If such a state is found, the model is consistent which means no constraint contradiction occurs.

B. *USE Monitor*

The runtime monitor in USE [6] is realized as a plugin. It currently supports the monitoring of applications running inside a Java virtual machine (JVM). The monitor requires a so-called platform aligned model (PAM) which specifies central aspects of a system to monitor. It is called platform aligned, because information about the implementation is needed within the model, e. g., package names or attribute names for association ends. The monitor can be attached to a running system at any time. After it is connected it takes a snapshot of the running system and maps instances inside the virtual machine to instances of classes of the PAM. The snapshot only contains instances of modeled classes, attributes and associations. Therefore, a snapshot can be seen as a subset of the central data of the running system. After this initial snapshot has been taken a user can examine static aspects of the system by checking structural constraints, e. g., specified multiplicities or invariants. Dynamic validation can be done by resuming the system which is monitored. After the application has been resumed by the monitor, the monitor reacts on several events coming from the virtual machine to keep track of changes and to be able to synchronize the snapshot with the running instance. When monitored operations (operations specified in the PAM) are called inside the running system, the monitor pauses the running system and validates the specified preconditions for the operation. If a precondition fails the user is notified and she can react on this violation by examining the failed precondition and the current system state. Analogously to normal testing, she has to decide whether the specified constraint is erroneous or a feature of the implementation. If no violation is encountered the execution is continued until the operation is returning or other monitored operations are called. When the operation is returning, the monitor pauses the execution again and checks the specified postconditions of the operation. At this point, a central benefit of reusing USE as an execution environment for models gets visible. USE records the system state before an operation was called which allows the validation of postconditions including the usage of the OCL-keyword @pre. In our previously published work [6] the monitor was controlled by simple shell commands inside of the USE shell. The approach was extended to a more intuitive user interface (see Fig. 4). This also allows finer controlled messages to the user and an elegant way to integrate model breakpoints into the monitor in the future.

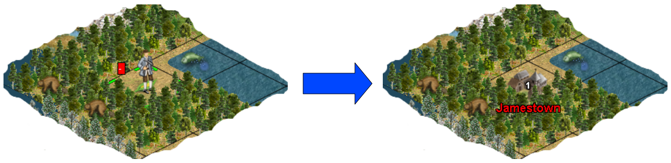
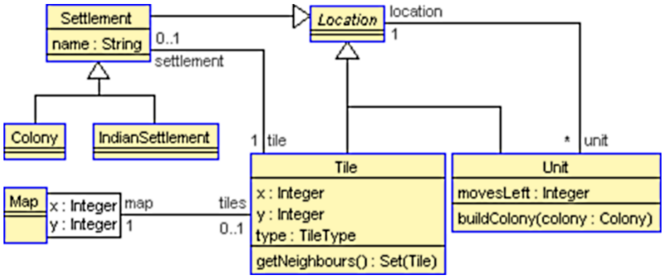


Fig. 2. Parts of a running FreeCol game

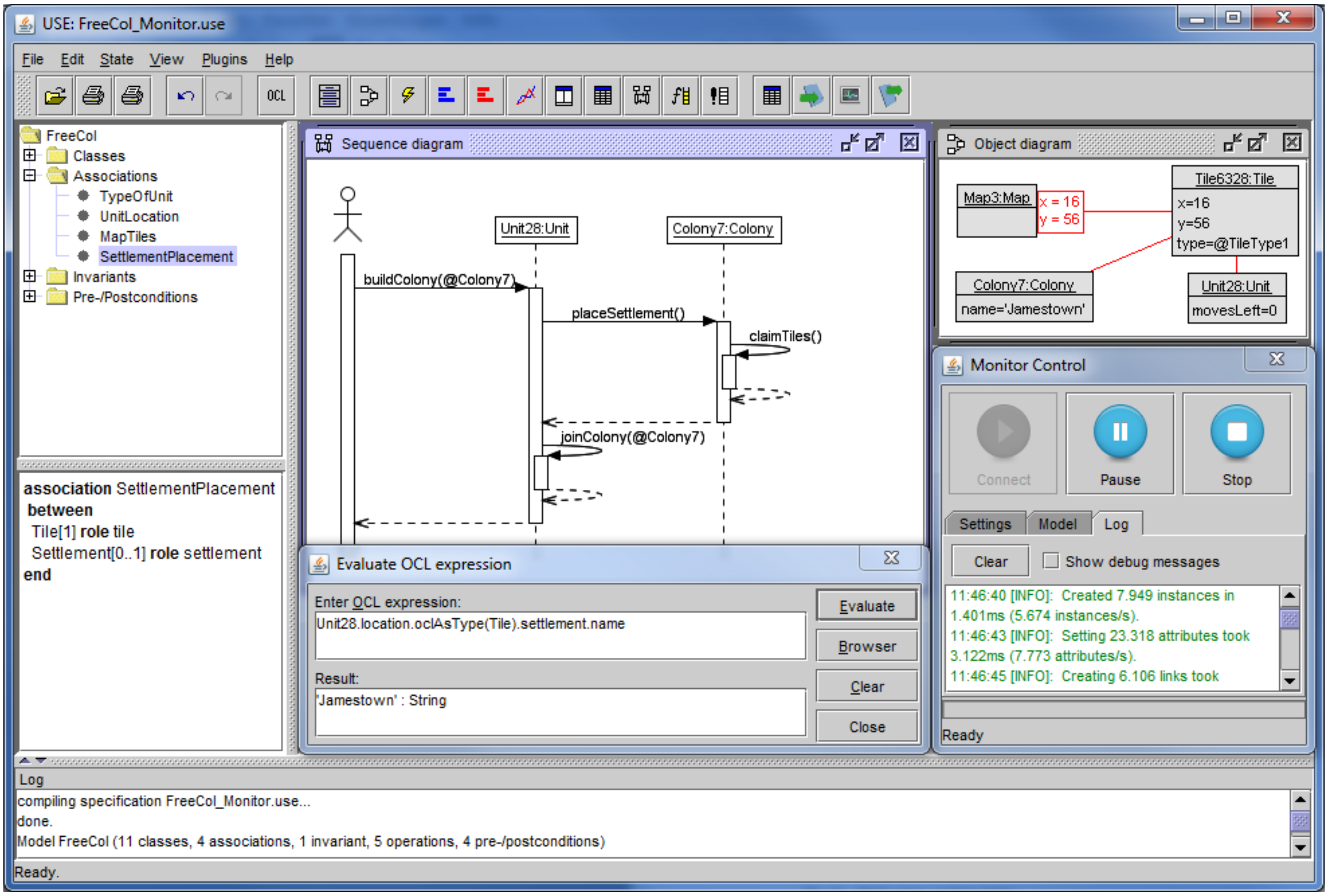


```
context Unit::buildColony(colony:Colony)
  pre movesLeft: self.movesLeft > 0
  pre noSurroundingColonies:
    self.location.oclIsKindOf(Tile) and
    self.location.oclAsType(Tile).
      getNeighbours()->forAll(t |
        t.settlement.isUndefined())
```

Fig. 3. PAM for FreeColonization and two assumptions

C. *Sample monitoring process*

To be able to compare the results of the semi-automatic PAM extraction process to a hand written PAM we reuse the example shown in [6]. In the example we build a PAM for the open source computer game *Free Colonization*. We concentrate on monitoring the execution of one central functionality of the game: the founding of colonies. The example game situation is shown in Fig. 2. The left part is the state before the founding of a colony, whereas the right part shows the state after founding the colony Jamestown. A PAM for the game with assumptions about the behavior formulated as pre- and postconditions is shown in Fig. 3. In addition to the pre- and postconditions introduced to the model, also a query operation `getNeighbours()` was added to the PAM to be able to reuse this expression. The presented preconditions in Fig. 3 state, that an operation call to `buildColony` is only valid, if the unit has moves left and there are no surrounding colonies. The screenshot of the USE system presented in Fig. 4 shows the result of attaching the monitor to FreeCol when the game is in the left state of Fig. 2 and monitoring the execution of the founding of Jamestown which results in the right state of Fig. 2. Parts of this state are shown as an object diagram in USE on the right upper side of the screenshot. The execution flow is shown in the center of the screenshot. Please note, we used a more detailed PAM including additional operations for the screenshot to make it more meaningful but we show only a fragment in Fig. 3.



```
association SettlementPlacement
 between
  Tile[1] role tile
  Settlement[0..1] role settlement
end
```

Enter OCL expression:

```
Unit28.location.oclAsType(Tile).settlement.name
```

Result:

```
'Jamestown' : String
```

```
11:46:40 [INFO]:  Created 7.949 instances in
1.401ms (5.674 instances/s).
11:46:43 [INFO]:  Setting 23.318 attributes took
3.122ms (7.773 attributes/s).
11:46:45 [INFO]:  Creating 6.106 links took
```

```
compiling specification FreeCol_Monitor.use...
done.
Model FreeCol (11 classes, 4 associations, 1 invariant, 5 operations, 4 pre-/postconditions)
```

Fig. 4. System state presented in USE while monitoring

D. Support for PAM extraction

In our previous work the platform aligned model was created manually by exploring the source code as shown in Fig. 3. Although finding the appropriate part of a large program remains a human task, it can be fairly supported by reverse engineering techniques. The main problem with an automatically extracted model is its size. Coping with huge models at runtime is challenging: they cause severe performance issues for modeling tools; and they hinder the visual observation and program understanding of the developer.

We approximate the PAM model by a reverse engineered class diagram. We extract the PAM of the whole application from the source code and export it as a USE model. The essential, important part of the model is achieved using further filtering. Thus we employ filtering at two levels:

- pre-filtering - unnecessary details are filtered out during the model building phase
- USE filtering - search/select the important part of the model in USE

Figure 5 contains the overview of the tools used for automatic extraction of PAM. Static analysis of source code is done by the Columbus Java analyzer [7]. The obtained program model is converted to a higher level, language independent object-oriented model. The existing Columbus tool-chain is extended with a new module, which computes the PAM and exports it to a USE model.

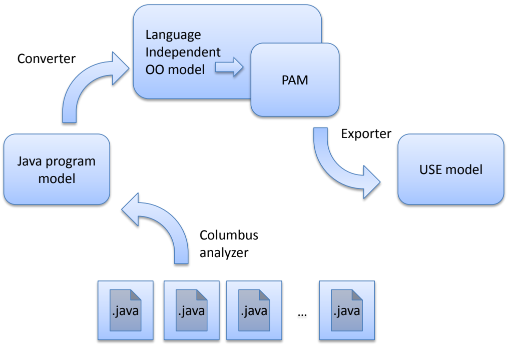
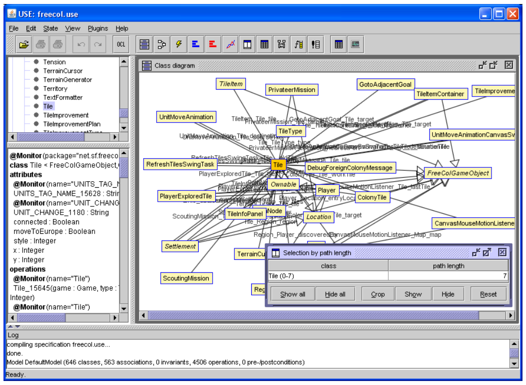


Fig. 5. Tool architecture of the PAM Extractor

During PAM extraction we obtain facts from the source code and convert them to a valid USE model. First, a base class diagram is built consisting of classes, attributes, methods, inheritance relationships and associations. Associations are extracted as suggested by Kollmann et al. [8]. The final model has to conform to several rules like source code traceability, concise and consistent naming of elements and unique navigability of associations. Source code elements at some points break the well-formedness rules of USE models, e.g. when an attribute is defined both in a base and in a descendant class. To overcome these problems, the names in the model are changed at several points - shortened or made unique by appending unique identifiers to names. The source code traceability of modified names is assured by name annotations in the USE model. Pre-filtering currently consists of dropping out attributes taking part in associations and filtering out Java library classes and their references (attributes, methods with library class parameters).

We validated our reverse engineering solution by extracting the model of the FreeCol program. The extracted USE model was filtered to be comparable to the model made previously by hand. In the USE system there are several possibilities to search classes and their neighbours; and to crop and hide appropriate classes to get a reduced model showing essential part of the application. A typical filtering step can be seen in Fig. 6: the immediate neighbours of selected class *Tile* are shown, while others are hidden.



```
Tension
TerrainCursor
TerrainGenerator
Territory
TextFormatter
Tile
TileImprovement
TileImprovementPlan
```

```
@Monitor(package="net.sf.freeco
class Tile < FreeColGameObject,
attributes
 @Monitor(name="UNITS_TAG_N
 UNITS_TAG_NAME_15628 : Strin
 @Monitor(name="UNIT_CHANG
 UNIT_CHANGE_1180 : String
 connected : Boolean
 moveToEurope : Boolean
 style : Integer
 x : Integer
 y : Integer
operations
 @Monitor(name="Tile")
 Tile_15645(game : Game, type :
Integer)
 @Monitor(name="Tile")
```

class	path length
Tile (0-7)	7

```
Log
compiling specification freecol.use...
done.
Model DefaultModel (646 classes, 563 associations, 0 invariants, 4506 operations, 0 pre-/postconditions)
Ready.
```

Fig. 6. Examining neighbours of class *Tile*

Figure 7 shows the automatically extracted model after filtering in USE. The main difference compared to Fig. 3 lies in the discovered associations. *Settlement* and *Tile* are associated in both directions, but there is also an additional association from the direction of *Tile* pointing to the owning settlement. Similar observations can be made with *Unit* and *Location* as well. Furthermore, *type : TileType* is an attribute of class *Tile* in the manual model, while it is generated as an association according to the rules of automatic model extraction.

Finally, using the generated model we successfully reproduced the same condition checking procedure as done previously on manually the created models.

III. Conclusions

We have presented a tool-chain that allows developers to monitor a Java application in form of a platform aligned model (PAM) enriched by OCL requirements. With an example we have shown that the tool-chain is capable of handling non-trivial applications with several hundred classes. As future work, larger case studies have to be carried out. In order to find key classes and to support the PAM discovery, concept location techniques could be applied. Furthermore, we think of (what we would call) 'model breakpoints' which permit a developer to force the application to pause at a certain point in the model, not on a specified line in the code. Model breakpoints may be employed in connection with particular conditions. Another line of research would be to incorporate traces in the approach so that certain operation call sequences can be monitored. Last but not least we could check to what extent the approach is applicable to other virtual machines like CLR (Common Language Runtime) for .NET languages.

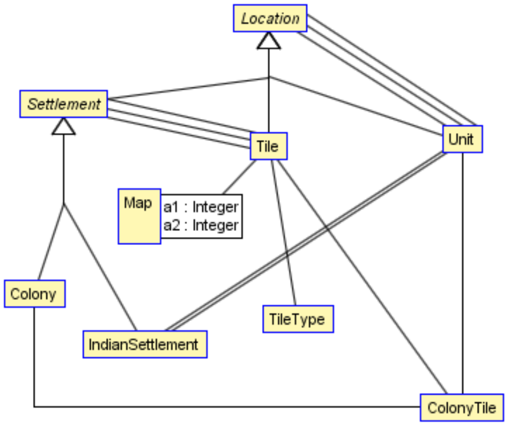


Fig. 7. Extracted USE model of the Freecol example

References

[1] *Object Constraint Language Specification Version 2.2*, OMG - Object Management Group, Feb. 2010. [Online]. Available: http://www.omg.org/spec/OCL/2.2

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

[3] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Ros, u, "An Overview of the MOP Runtime Verification Framework," *International Journal on Software Techniques for Technology Transfer*, 2011.

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

[5] M. Gogolla, M. Kuhlmann, and L. Hamann, "Consistency, Independence and Consequences in UML and OCL Models," in *Proc. 3rd Int. Conf. Test and Proof (TAP'2009)*, C. Dubois, Ed. Springer, Berlin, LNCS 5668, 2009, pp. 90–104.

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

[7] L. Schrettner, L. J. Fülöp, R. Ferenc, and T. Gyimóthy, "Visualization of software architecture graphs of java systems: managing propagated low level dependencies," in *Proceedings of the 8th International Conference on Principles and Practice of Programming in Java, PPPJ 2010, Vienna, Austria*, 2010, pp. 148–157.

[8] R. Kollmann and M. Gogolla, "Application of the UML Associations and Their Adornments in Design Recovery," in *Proc. 8th Working Conference on Reverse Engineering (WCRE'2001)*, P. Aiken and E. Burd, Eds. IEEE, Los Alamitos, 2001.

Acknowledgment: The work of László Vidács was supported by the DAAD.