OCL-Based Runtime Monitoring of Applications with Protocol State Machines

Lars Hamann, Oliver Hofrichter, and Martin Gogolla

University of Bremen, Computer Science Department
Database Systems Group, D-28334 Bremen, Germany
{lhamann,hofrichter,gogolla}@informatik.uni-bremen.de

Abstract. This paper presents an approach that enables users to monitor and verify the behavior of an application running on a virtual machine (like the Java virtual machine) at an abstract model level. Models for object-oriented implementations are often used as a foundation for formal verification approaches. Our work allows the developer to verify whether a model corresponds to a concrete implementation by validating assumptions about model structure and behavior. In previous work, we focused on (a) the validation of static model properties by monitoring invariants and (b) basic dynamic properties by specifying pre- and postconditions of an operation. In this paper, we extend our work in order to verify and validate advanced dynamic properties, i. e., properties of sequences of operation calls. This is achieved by integrating support for monitoring UML protocol state machines into our basic validation engine.

1 Introduction

When one faithfully follows the Model-Driven Development (MDD) paradigm, abstract representations of all artifacts, in particular of code, are needed in form of models. Model-like descriptions can be used as central parts in the software development process and are considered to be a promising paradigm for effective software production. Models can be employed in all development phases and for different purposes. Consequently and despite all justified criticism, the Unified Modeling Language (UML) is playing a pivotal role as a modeling language. Nearly every software engineer understands at least the UML core concepts, while other more specialized modeling languages first need to be explained from the scratch. This central role of the UML can also be observed by looking for transformation approaches from UML to more formal and specialized languages or tools such as the Alloy [24] language, SAT [25] or model checkers [19].

When using UML models for abstractions of concrete software systems, model quality is important. It has to be ensured that the developed models correspond to the implementation to be abstracted from. Otherwise formal quality assurance techniques would verify some disconnected abstract model and not the concrete implementation. This is especially true, if the implementation is not fully generated from the model and finalized by a developer. This is currently the most common case.

In [17] simulation of the model is proposed in the overall process of model checking. The process is shown in Fig. 1 which is adapted from [17, p. 8]. Our contribution and extension to the process is shown in the parts having a grey background.

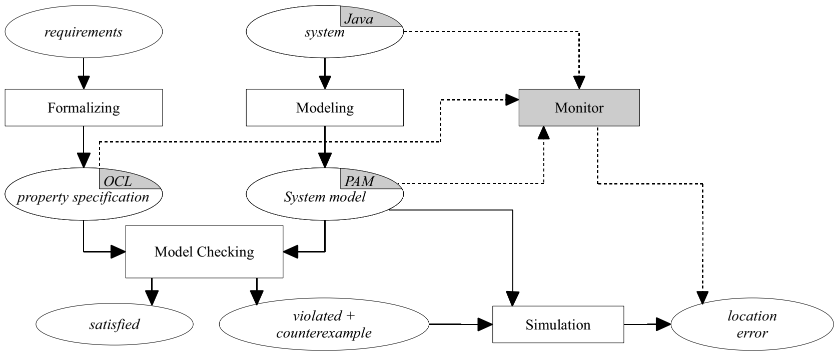

Fig. 1. Monitoring in the context of Model Checking (c.f. [17])

In our approach, we do not only simulate the model. We combine the system model with the implementation (the system) in order to be able to detect mismatches between the implementation, the system model and the property specification. We do so while executing the actual implementation. As systems we consider applications running inside a virtual machine, such as Java application running inside the Java Virtual Machine (JVM). Our system model will be defined as a UML class model extended by UML protocol state machines [20] and augmented with property specifications resp. assumptions formulated as OCL (Object Constraint Language) [21] state invariants and OCL operation pre- and postconditions. Since the elements of this system model need to be identified by our monitor, the system model needs to be aligned to the implementation. We call such a model a platform aligned model (PAM). We connect these components with a monitor in order to verify assumptions about the components at runtime. Our monitor can be started at any time that the concrete system, i. e., the Java application, is running. As an extension to our work presented in [15] and [16], we show how a state machine extension of the employed validation engine can be used without modifying our monitor component. Here, we show how UML protocol state machines (psms, singular psm) can be used to validate the correct sequence of operation calls, i. e., a protocol definition for a given class. We will further discuss some threads to validity which have to be considered when using a monitor approach like ours.

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 gives an overview on the integration of protocol state machines into our validation engine USE [11]. Section 4 explains the employment of protocol state machines in combination with our monitoring approach by means of a middle-sized case study applied in our tool USE. Section 5 discusses related work. The paper ends with a conclusion and ideas for future work.

2 Monitoring

In this section we explain our monitoring approach. A more detailed description can be found in [15]. The main idea of our approach is to monitor a running implementation of a system and to extract a more abstract representation of the current system state into a validation engine. We call this abstract representation a *snapshot* of the system under monitoring (SUM), because in general it is a small subset of the artifacts of the running system. Since we want to focus only on central parts of the implementation we leave out unimportant parts. The basis for this snapshot is a model which is more abstract than the implementation, e. g., by defining associations which are not present in programming languages, but specific enough to be able to find relevant parts inside the SUM, e. g., by specifying concrete package names. Because of this alignment between the most specific platform model, e. g., byte code and platform independent models we call this model level *platform aligned model (PAM)*.

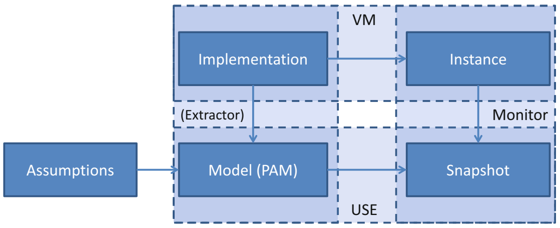

Fig. 2. Overview of the monitoring approach

As shown in Fig. 2 the PAM is enriched with assumptions about the running system. These assumptions are verified during the monitoring process by our validation engine USE. In order to be able to verify assumptions specified in a model USE needs an instance (i. e., objects and links) of it. In the monitoring context we call this instance a snapshot. Figure 2 shows this relation at the bottom. The monitor ensures, that the instance required by USE is a valid snapshot of the monitored instance inside the virtual machine. The virtual machine itself as shown at the top of the figure uses the implementation and an instance, i. e., the (heap) memory, stack, stack pointer, etc. of a running program. The PAM can be defined in several ways. For example, it can be step-wise refined when developing a system or it can be extracted by using reengineering techniques as shown in [16]. Furthermore, it can be generated when using model driven development.

Using modern virtual machine implementations like the JVM or the CLR of Microsoft .NET allows our monitor to use a rich pool of debugging and profiling interfaces. For example, the Java Platform Debugger Architecture[22] enables third party tools to easily access applications running inside a local or remote virtual machine. An important part of this interface is the possibility to retrieve information about instances of a specific type. This is used as an entry point for our monitoring approach described next.

First, the validation engine needs to be configured with the corresponding PAM and the SUM needs to be started. Next, the monitor needs to be connected to the running system. If the startup of a SUM is important, the user can also start the application with specific parameters, so that it suspends directly when started and is resumed only if the monitor signals this to the application. When the monitor is connected after the application is already running, the monitor creates a snapshot of the current system state. The following descriptions of the steps to create this abstract snapshot are explained in more detail in [15].

1. For all classes in the PAM which can be matched to an already loaded class in the VM, all existing instances of them are mapped to newly created instances of the platform aligned model.
2. For each created instance in the previous step the values of the attributes defined in the PAM are read. This step includes a mapping for values of primitive types to built-in OCL types, e. g., String and Real (c. f. [28]). Attribute values with a type of a class defined in the PAM need to be mapped using the mapping created in the first step.
3. For all associations in the PAM, links are created between corresponding instances.
4. By using the current stack-trace of the monitored system the current operation call sequence relevant to the monitored elements can be rebuilt. For this, the deepest operation call to a monitored operation (an operation specified in the PAM) on the call stack acts as an entry point for the following monitored operations on the call stack.

After such a snapshot has been constructed, the monitor needs to register to several events that occur in the VM in order to keep the snapshot synchronized with the running system and to allow a dynamic monitoring of the SUM. Currently our monitor makes use of the following breakpoint and watchpoint locations:

1. At class initialization to allow the registration of all other breakpoints. This ensures, that classes which were not loaded while taking the snapshot are also monitored.
2. At constructors of monitored classes. This allows the monitor to keep track of newly created instances and therefore enables an incremental construction of the system state in contrast to always construct a new snapshot of the running system when needed.
3. At the start of a monitored operation. This enables the monitor to validate preconditions at runtime and to follow the call sequence.
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.

5. When a monitored attribute is modified. A monitored attribute might be an attribute or association end inside of the PAM.

Monitoring an application in the presented way in combination with our validation engine USE allows a user to monitor the validity of UML constraints like multiplicities or composition properties, invaraints, pre- and postconditions *without* the need to modify the source code of the application or to use special bytecode injection mechanism. In addition, without changing the monitor component, improvements made to the validation engine can be used. For example after adding support for protocol state machines to USE, as described next, only the PAMs of the monitored systems needed to be extended to allow a more detailed monitoring of call sequences. Without the use of protocol state machines, only a very small part of a call sequence could be validated in one step, because OCL only allows access to the state just before an operation was called. Using protocol state machines it is possible to validate operation call sequences of arbitrary length.

3 Protocol State Machines in USE

The UML specifies two kinds of state machines: behavioral and protocol state machines [20]. As the name suggest, the former kind is used to specify the behavior of UML elements including actions attached to transitions to specify changes inside a system while taking a transition. The latter one specifies the allowed call sequences of a protocol. In USE we added support for protocol state machines in the context of a class. Following the general idea of USE, we start with a small well-defined subset of the many features for UML state machines. In the following we describe this implemented subset and its semantics.

First of all, all state machines in USE are flat, i. e., they have only one region and no composite states. They have only a single initial and a single end state. All other states are proper states and no pseudo states, which means that there are no forks or joins. States can have a state invariant which needs to be valid if a given psm instance is in the corresponding state. The context of a psm instance and also for the state invariant (accessed by using `self` in an OCL expression) is the instance of the context class of the psm which owns the psm instance. For the initial state only an unnamed transition or a transition with the event `create` is allowed as an outgoing transition. An initial state has no incoming transitions while an end state has no outgoing transitions. The transitions between states specify the valid call sequences of operations for the context class. As described in the UML the protocol state transitions between states consist of three parts:

1. the referred operation (`op`),
2. an optional guard (`G`), i. e., a precondition and
3. a postcondition (`PC`) which is also optional.

In an state machine diagram the transitions are labeled using the following schema: $\xrightarrow{\texttt{[G] op()/ [PC]}}$. A state can have multiple outgoing transitions that refer to the same operation. To be still able to choose a single transition the guard, post condition and state invariant of the target state for all transitions referring to the same operations are considered by USE. In some situation the usage of all this information still leads to multiple possible transitions. When USE encounters such a situation it reports an error to the user.

When an operation on an object whose class defines at least one psm is called, the selection of the transition to be taken for each psm is done in the following way. First, it is checked if the operation call needs to be ignored, i. e., no transition must be taken. This is the case if

- none of the transitions inside the protocol state machine covers the called operation (see [20, p. 545]) or
- the psm is not in a stable state, i. e., a transition is currently active.

If the operation cannot be ignored it is checked

- if at least one outgoing transition of the current state is enabled, i. e., the state has one or more outgoing transitions which refer to the called operation while having a valid precondition.

All enabled transitions are saved as possible transitions which could be taken after the operation call is completed. When the called operation finishes its execution, for all possible transitions the postcondition and the state invariant of the target state are validated. If only one transition fulfills the postcondition and the state invariant the transition is taken. Otherwise an error is reported which also explains if either no transition could be taken or multiple transitions would be possible.

3.1 State Determination

One benefit of our monitoring approach is the possibility to connect to a monitored system at any time. While this allows a SUM to run without overhead until the monitoring starts, this ability leads to some issues to be considered. One major problem is the lack of information of previously called operations, so that all protocol state machine instances are in an undefined state. To allow a correct monitoring of psms it is important to determine the correct states of all psm instances. To be able to determine the states after an initial snapshot has been taken we use state invariants. These state invariants need to be well-defined because otherwise the snapshot would be in an unsound state. For example, all psm instances should be in a given state after the state determination check. In this context well-defined means that the state invariants should be independent of each other, i. e., at any state only one state invariant evaluates to true for every instance referring to the psm.

When using complex state invariants the task of verifying the independence of state invariants can be accomplished by using automatic model finding techniques. These are similar to the one presented in [13] which allows a user to show the independence of invariants. In [13] the independence of invariants is slightly different form the independence of state invariants we want to achieve. In [13] an invariant is defined as independent if it cannot be removed without loss of information meaning, there exists at least one system state where this single invariant is violated. For the independence of state invariants required for the state determination, we consider state invariants as independent if for all system states only a single state invariant is fulfilled.

Formally, given the set of all possible system states $\sigma(M)$ of a Model M and the invariants $i_1, \ldots, i_n$ the independence of an invariant i_k is defined in [13] as

$$\exists \sigma \in \sigma(M)(\sigma(i_1) \wedge \cdots \wedge \sigma(i_{k-1}) \wedge \sigma(i_{k+1}) \wedge \cdots \wedge \sigma(i_n) \wedge \neg\sigma(i_k))$$

whereas in this work the independence of state invariants $i_1, \ldots, i_n$ for a single psm is defined as

$$\forall \sigma \in \sigma(M)(\sigma(i_k) \Rightarrow \neg\sigma(i_1) \wedge \cdots \wedge \neg\sigma(i_{k-1}) \wedge \neg\sigma(i_{k+1}) \wedge \cdots \wedge \neg\sigma(i_n))$$

However, the same validation techniques apply, but as the universal quantification indicates, a full verification requires a complete search through all possible system states, which implies the well-known state space explosion problem and is therefore not a trivial task and we restrict ourselves to checking occurring test cases.

4 Case Study

In this section we apply our extensions to the monitoring approach to the public available, mid-sized application we used in [15]. The case study will demonstrate the advantages of our approach.

- Assumptions about a running implementation can be validated without the need to modify the source code.
- The state of an implementation can be examined in an abstract way to discover inconsistencies or design decisions.
- Using protocol state machines the correct usage of the defined protocol of a class can be validated.
- Concrete usage scenarios can be visualized by means of a sequence diagram.

This will be exemplified by the following case study using an open source computer game called *Free Colonization*[1] or in short *FreeCol*. It is a modern Java-based implementation of the 1994 published game *Sid Meier's Colonization*[2]. 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

[1] Project website: `http://www.freecol.org`

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

colonies, trade goods, etc. Figure 3 shows an example state transition of a running game. One unit (i. e., a pioneer) is placed in the center of the shown map on the left side and is surrounded by several different tile types. The right map shows the game state after the pioneer has build a new colony called Jamestown. The sketched state machines displayed below the two maps exemplify our new contribution. We want to be able to monitor the transition of the pioneer state from one state before she or he built a colony to another state after she or he joined the colony (note, that this is a single step in the game).

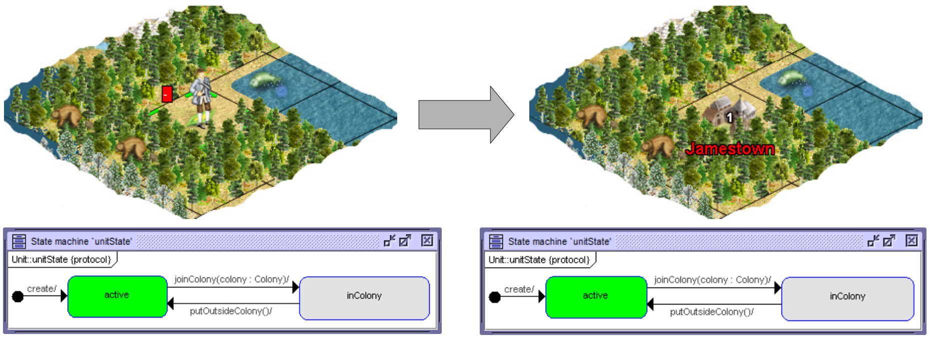
Jamestown
State machine `unitState'
Unit::unitState {protocol}
create/
active
joinColony(colony : Colony)/
putOutsideColony()/
inColony
State machine `unitState'
Unit::unitState {protocol}
create/
active
joinColony(colony : Colony)/
putOutsideColony()/
inColony

Fig. 3. Sample game situation in FreeCol

To be able to monitor this transition, we extended the PAM presented in [15] in a step-wise manner. First we added the enumeration `UnitState` to our PAM and defined a new attribute `state:UnitState` to the class `Unit` as shown in Fig. 4. The presence of this attribute simplified the definition of the state invariants as we will see later.

For our purpose the class diagram shown in Fig. 4 with an overall of 14 classes is detailed enough. When compared to the 551 classes which are present in version 0.9.2 of FreeCol we used for the monitoring this illustrates that the PAM for an application only needs to represent a small subset of the monitored implementation. Because we focus on state transitions we do not show any constraints defined for the PAM. Examples can also be found in [15].

Except for one case, we modeled attributes of a Java class which use a class present in the PAM as associations. The exception is the attribute `Tile::type` which reduces the number of links in an object diagram, but still allows to directly see that tiles differ in their type. For a first definition of a psm which monitors the entrance and the exit of a colony for a unit, we only need to consider the class `Unit` and its operations `joinColony(aColony:Colony)` and `putOutsideColony()` in combination with the attribute `state:UnitState`. These elements are present in the concrete implementation of the class `Unit` and can directly be monitored. The enumeration `UnitState` defines nine different enumeration literals which express different states of a unit. Since we are only interested in the state `IN_COLONY` and do not consider the other states we

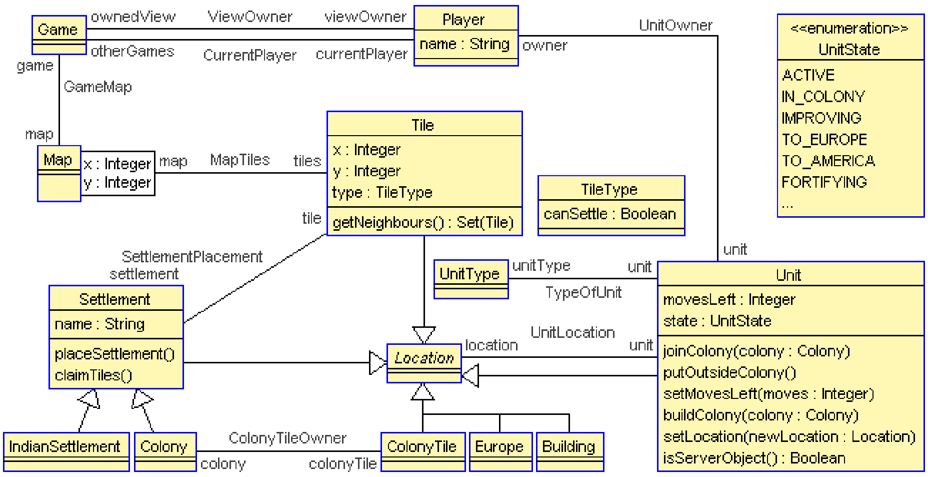

Fig. 4. Platform aligned model

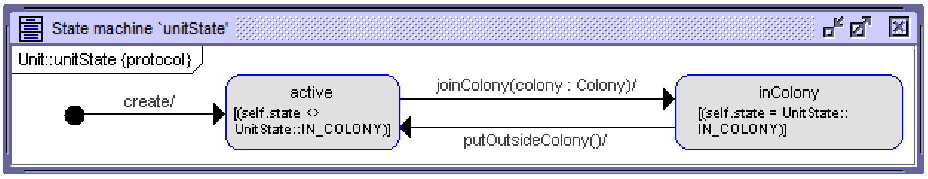
State machine `unitState'
Unit::unitState {protocol}
create/
active
[(self.state <> UnitState::IN_COLONY)]
joinColony(colony : Colony)/
putOutsideColony()/
inColony
[(self.state = UnitState::IN_COLONY)]

Fig. 5. Protocol state machine for the class Unit

can specify a protocol machine with two states. One for the state `IN_COLONY` and one for all other game states. Our assumption about the protocol of the class `Unit` is that an operation call to `putOutsideColony()` is only valid after the operation `joinColony()` has been called on the same object sometime before.

To be able to set the correct state of a monitored instance we need to specify state invariants for these two states. As stated earlier, the presence of the attribute `state` for the class `Unit` simplifies this task, because we only need to check the value of the attribute to determine the current state after a snapshot has been taken. Therefore, the state invariant for the psm state `inColony` is `self.state = UnitState::IN_COLONY` and the other state invariant only changes the comparison from equal to not equal. Given the previously expressed assumptions, this leads to a psm which has two states and two transitions leaving out the transition for the creation. This psm is shown in Fig. 5. A `Unit` object starts in the state `active` after it is created and enters the state `inColony` when the operation `joinColony(colony:Colony)` was executed. If the operation `putOutsideColony()` is called the state changes back to `active`. Any other operation call to a unit instance is ignored as described in the UML specification for operation not mentioned in a psm. This means, the psm only allows a state change when one of the two operations is called.

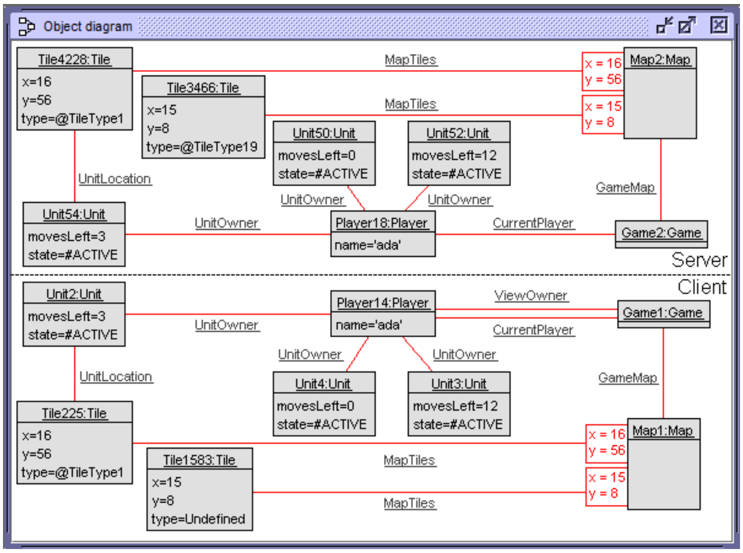

Fig. 6. Parts of the snapshot taken at runtime

Figure 6 shows the relevant part of the snapshot after connecting to the running game when it is in the game state shown on the left of Fig. 3 as an object diagram. The overall snapshot consists of nearly 6000 objects and 4000 links which makes it impossible to manually extract an informative object diagram. USE allows a user to select objects which should be shown or hidden in an object diagram by using several features. Two useful ones are the selection by an OCL expression and the selection of related objects by path length (see [12] for more information). The shown part of the snapshot is divided into two parts, which are important while validating the assumptions about the state transitions. Because we monitored a single user game on a single machine the instance of the game contains both, the data used by the game server and the client. By looking at the instances `Tile3466` on the server side and `Tile1583` on the client side one can see that the server part has more information about the game than the client part. Both instances represent the same tile on a map, because their positions are equal, but the client instance does not know of what type the tile is. To be able to determine if an object belongs to the server or client side we also monitored the class game with the association `ViewOwner`. If a game object is not linked to a player by this association it is the server game. The equivalent OCL expression (`self.owner.ownedView->isEmpty()`) is used as a body for the operation `isServerObject()` of the class `Unit`. This operation is marked as a query operation and is therefore ignored by the monitor. The object diagram further shows the owned units of the player named ‘ada’ and the object for the tile on which we want to build a colony (`Tile4228` resp. `Tile225`).

After taking this initial snapshot, the states of the protocol state machines for the existing unit objects need to be determined. This can be done by a single command in USE which also informs the user about objects for which the psm instance could not be set to a single state. This happens, if no state invariant or multiple state invariants evaluate to true w. r. t. the given snapshot. Because this state determination is a common task after a snapshot has been taken, the monitor plugin can automatically execute the state determination after the construction of a snapshot. After the states have been determined the states of the relevant units of the snapshot are as expected (`active`). After resuming the game and building the new colony *Jamestown* we get a valid sequence of operation calls which can be seen in the monitored sequence diagram shown in Fig. 7. We observed, that the execution of the operation `joinColony()` indeed leads to the attribute value `IN_COLONY` of the attribute `Unit::state`, because no violation of a transition is reported.

To get further information about our assumptions we can instruct USE to validate the current state invariants of all psm instances. After the validation of our current snapshot USE reports an error for the psm instance of the client object of the unit which has built the colony. This is due to the fact that the operation `buildColony` is only called on the server object and only the new values are transfered to the client object. Therefore, USE did not execute a transition from the source state `active` to the target state `inColony` for the client unit but monitored the change of the attribute `state` to `IN_COLONY`. Now, the new attribute value violates the state invariant of the state `active`.

Because the separation of the client and server objects seems to be a valid design decision we can ignore these violations and continue the monitoring process to retrieve further information about the validity of our assumptions. To test the defined protocol we use another unit and let it join and exit the colony. While executing this scenario another issue arises because entering an existing colony, i. e., a unit only enters a colony without building it before, does not lead to an operation call to `joinColony()`. Instead, only `setLocation()` is called which is not handled by the psm and therefore does not execute a transition keeping the psm instance in the state `active`, but the attribute value of the runtime instance is set to `IN_COLONY` which violates the state invariant of the state `active`.

Using this information a user of the monitor needs to decide where the error is located: in the implementation or in the PAM. For our example, we assume that the PAM needs to be modified although it seems to be an unsound usage of the `Unit` class. This assumption is backed by the fact, that the developers of FreeCol refactored this part of the game in newer releases. If we want to adapt our psm to the last discovered facts, we need to handle the client server separation and the additional operation calls. The modified psm is shown in Fig. 8. The additional operation `setLocation(newLocation:Location)` leads to two new transitions in the psm. Both transitions have as their source state the state `active` but differ in their target and guard. If the new location is of type `ColonyTile`, which represents special tiles related to a colony, the new state after the execution is `inColony` otherwise the state does not change. Interestingly, when a `Unit` object leaves a colony this leads always to a call to `putOutsideColony()`.

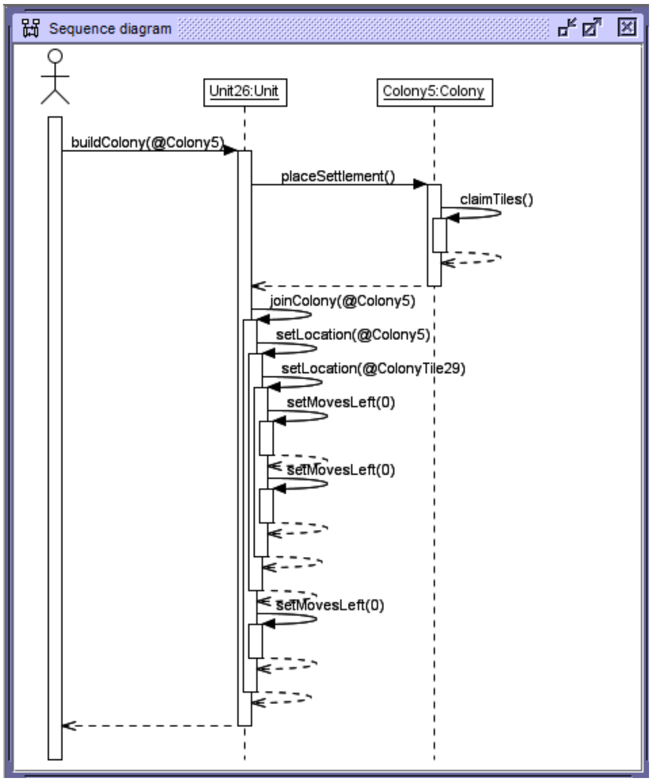

Fig. 7. Sequence diagram of the monitored execution

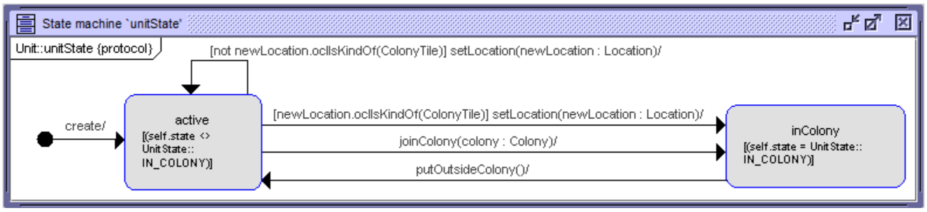
State machine `unitState'
Unit::unitState {protocol}
[not newLocation.oclIsKindOf(ColonyTile)] setLocation(newLocation : Location)/
create/
active
[(self.state <> UnitState::IN_COLONY)]
[newLocation.oclIsKindOf(ColonyTile)] setLocation(newLocation : Location)/
joinColony(colony : Colony)/
putOutsideColony()/
inColony
[(self.state = UnitState::IN_COLONY)]

Fig. 8. Extended PSM for the class Unit

However, the problem how to differentiate the server and client objects still exists. When taking a snapshot all state invariants should be independent to allow a valid determination of the current state. If we would introduce a new state for client objects with the state invariant `not self.isServerObject()` and add a new conjunction `self.isServerObject()` to the two existing state invariants the state determination would work, because each instance can be mapped to a single state. Either it is a client object or it is a server object and the two server states are independent because of the different comparison operators. The problem with this approach is the dynamic monitoring after existing instances are read. A new instance would be in the state `active` which is now only defined for a server object due to the new conjunction. If a new client instance is created this would violate the state invariant. Further, the new client instance would never change the state because none of the monitored operations is called for client objects as described before. Using an implies condition would violate the independence of the two server states because for client objects both invariants would be fulfilled.

A simple solution would be to add a transition which covers the operation that specifies the new instance to be a client instance. However, FreeCol does this inside of the constructor which is represented as the `create` transition and the UML explicitly forbids multiple outgoing transitions for the initial state and also no post condition on it. If it was allowed the distinction could be done using postconditions on two different create transitions. Another solution for this is to use a change event on a transition. By specifying the change expression `not self.isServerObject()` a psm instance can move to a specify client state. A drawback of this solution is the relative high calculation cost of such change events. Because a change expression can generally access every object or property of the current snapshot all currently valid transitions with change events would need to be checked after every change in the snapshot. The costs can be reduced by using special analysis algorithms which calculate the change expressions that need to be checked after a change as presented in [7]. Currently, such a mechanism is not present in USE, but could be integrated in the future. For now, we need to ignore state violations of the client objects. Note, that the validation of the correct transitions for the server objects still works.

When using this modified psm all scenarios described above lead to the expected changes of the psm states. Beside the manual execution of observed game situations the presence of computer controlled players in the game can be used as a test driver. As with the manual play all analyzed operations are also used by computer controlled players. We used this to strengthen our PAM.

5 Related Work

In our previous work we focused on the runtime verification of static properties (like multiplicity constraints and invariants) of an application running on a virtual machine [15]. Different approaches for checking information extracted from a running system for certain properties exist. [10] and [2] make a comparison between these approaches. According to [10] most constraint validation techniques for Java are based on the design-by-contract-principle introduced by the Eiffel programming language. In contrast to our approach, the approaches compared to each other in [2] require a full access to the source code of the system under monitoring. The Java Modeling Language (JML) is appropriate both for formal verification and runtime assertion checking [18].

In this paper, we extended our validation engine by support for UML protocol state machines in order to be able to verify and validate dynamic sequences of operation calls. Our approach applying protocol state machines differentiates from approaches which are based on the usage of regular expressions. Such an approach is presented in [5]. It enables programmers to define parameterized runtime monitors. For this purpose a temporal ordering over breakpoints, which are used for debugging purposes by programmers, is introduced. The temporal ordering is defined by regular expressions. Another approach uses tracematches for runtime verification [6]. As the previous approaches, this one is also based on regular expressions.

A UML protocol state machine as used in our approach is different from regular expressions through the information of transitions: protocol state machines provide the possibility to specify an initial condition (guard) under which an operation can be called. This possibility makes protocol state machines more powerful than regular expressions. The authors of [23] present an approach which applies UML protocol state machines to produce class contracts. For this purpose they define the structure and the semantics of UML protocol state machines.

With 'ocl2j' a tool exists which allows to enforce OCL constraints in Java through translating OCL expressions into Java code [9]. An analog approach is presented e. g. in [14]. From the authors runtime verification approach the tool 'INVCOP' has arised. The Dresden OCL toolkit makes available two distinctive approaches for OCL-based runtime verification [8]. While the 'generative' approach is based on the generation of AspectJ code, the 'interpretative' approach integrates the Dresden OCL2 Interpreter into a runtime environment in order to interpret OCL constraints.

In [4] the monitoring of state machines is focused while the usage of OCL is relinguished. With the 'aspect oriented approach', the 'listener approach' and the 'debugging approach', the authors describe three possibilities to extract runtime models.

To synchronize a running system with a runtime model the authors of [26] use 'synchronizers'. Thus the system can be changed immediately when the model has been updated and the model can be immediately adapted if the system progresses.

Java PathFinder (JPF) is a runtime verification and testing environment for Java developed at NASA Ames Research Center [27]. JPF is based upon a special Java Virtual Machine which is called from a model checking engine included in JPF. The authors of [1] present JPF-SE, a symbolic execution extension to JPF. The framework Polyglot has been integrated with the Java PathFinder [3]. Polyglot enables the execution of multiple variants of statecharts including UML statecharts and the verification of their models against properties. It uses an intermediate representation which is translated from a range of modeling tools. The intermediate representation is used to generate Java code representing the structure of a statechart which is analyzed by applying JPF.

6 Conclusion

We have presented an extension to our approach for monitoring assumed properties in form of OCL constraints for a running Java application. Based on this approach, which takes advantage of the powerful features of the Java virtual machine, we have added support for protocol state machines to the underlying validation engine. This allows us to specify assumptions not only formulated as class invariants or operation contracts, but also as state invariants. By using a protocol state machine, more knowledge about the history of an object is available because of the recording of states. We have shown that the definition of state invariants is important for our approach in order to determine the correct states of an object when connecting to a running system without the information about previous operation calls. We explained our work by a non-trivial example of an open-source game.

As future work we want to extend the support for protocol state machines within our validation engine. One major improvement would be the support for change events. To be applicable in practice, an efficient implementation is needed which considers only the transitions with an effective change event. A more detailed study of similar approaches, for example, based on aspect-orientation or approaches considering the Java Modeling Language (JML) as a target language, might introduce alternative features and our monitor could be improved in various directions. For example, one could consider abstract model breakpoints, which are configurable by the user or by extended information about elements that are only present within the running system. Last, but not least, comprehensive case studies must give more feedback about the applicability of our work.

References

1. Anand, S., Păsăreanu, C.S., Visser, W.: JPF–SE: A Symbolic Execution Extension to Java PathFinder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 134–138. Springer, Heidelberg (2007)
2. Avila, C., Sarcar, A., Cheon, Y., Yeep, C.: Runtime Constraint Checking Approaches for OCL, A Critical Comparison. In: SEKE (2010)
3. Balasubramanian, D., Pasareanu, C.S., Whalen, M.W., Karsai, G., Lowry, M.R.: Polyglot: modeling and analysis for multiple Statechart formalisms. In: Dwyer, M.B., Tip, F. (eds.) ISSTA, pp. 45–55. ACM (2011)
4. Balz, M., Striewe, M., Goedicke, M.: Monitoring Model Specifications in Program Code Patterns. In: Proc. of the 5th Int. WS Models@run.time, pp. 60–71 (2010)
5. Bodden, E.: Stateful breakpoints: a practical approach to defining parameterized runtime monitors. In: ESEC/FSE 2011. ACM, New York (2011)
6. Bodden, E., Hendren, L.J., Lam, P., Lhoták, O., Naeem, N.A.: Collaborative Runtime Verification with Tracematches. J. Log. Comput. 20(3), 707–723 (2010)
7. Cabot, J., Teniente, E.: Incremental integrity checking of UML/OCL conceptual schemas. Journal of Systems and Software 82(9), 1459–1478 (2009)
8. Demuth, B., Wilke, C.: Model and object verification by using Dresden OCL. In: Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice, Ufa, Russia, pp. 687–690 (2009)
9. Dzidek, W.J., Briand, L.C., Labiche, Y.: Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 10–19. Springer, Heidelberg (2006)

10. Froihofer, L., Glos, G., Osrael, J., Goeschka, K.M.: Overview and Evaluation of Constraint Validation Approaches in Java. In: Proc. of ICSE 2007, pp. 313–322. IEEE Computer Society, Washington, DC (2007)
11. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Computer Programming 69, 27–34 (2007)
12. Gogolla, M., Hamann, L., Xu, J., Zhang, J.: 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)
13. Gogolla, M., Kuhlmann, M., Hamann, L.: Consistency, Independence and Consequences in UML and OCL Models. In: Dubois, C. (ed.) TAP 2009. LNCS, vol. 5668, pp. 90–104. Springer, Heidelberg (2009)
14. Gopinathan, M., Rajamani, S.K.: Runtime Monitoring of Object Invariants with Guarantee. In: Leucker, M. (ed.) RV 2008. LNCS, vol. 5289, pp. 158–172. Springer, Heidelberg (2008)
15. Hamann, L., Gogolla, M., Kuhlmann, M.: OCL-Based Runtime Monitoring of JVM Hosted Applications. In: Proc. WS OCL and Textual Modelling. ECEASST (2011)
16. Hamann, L., Vidács, L., Gogolla, M., Kuhlmann, M.: Abstract Runtime Monitoring with USE. In: Proc. CSMR 2012, pp. 549–552 (2012)
17. Katoen, J.P., Baier, C.: Principles of Model Checking. MIT Press (2008)
18. Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the design of JML accommodates both runtime assertion checking and formal verification. Sci. Comput. Program. 55(1-3), 185–208 (2005)
19. Moffett, Y., Beaulieu, A., Dingel, J.: Verifying UML-RT Protocol Conformance Using Model Checking. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 410–424. Springer, Heidelberg (2011)
20. UML Superstructure 2.2. Object Management Group (OMG) (February 2009), `http://www.omg.org/spec/UML/2.2/Superstructure/PDF/`
21. Object Constraint Language 2.2. Object Management Group (OMG) (February 2010), `http://www.omg.org/spec/OCL/2.2/`
22. Oracle: Java^{TM}Platform Debugger Architecture - Structure Overview (2011), `http://download.oracle.com/javase/6/docs/technotes/guides/jpda/architecture.html`
23. Porres, I., Rauf, I.: From Nondeterministic UML Protocol Statemachines to Class Contracts. In: Int. Conf. on Software Testing, Verification, and Validation, pp. 107–116. IEEE Computer Society, Los Alamitos (2010)
24. Shah, S.M.A., Anastasakis, K., Bordbar, B.: From UML to Alloy and Back Again. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 158–171. Springer, Heidelberg (2010)
25. Soeken, M., Wille, R., Drechsler, R.: Encoding OCL Data Types for SAT-Based Verification of UML/OCL Models. In: Gogolla, M., Wolff, B. (eds.) TAP 2011. LNCS, vol. 6706, pp. 152–170. Springer, Heidelberg (2011)
26. Song, H., Huang, G., Chauvel, F., Sun, Y.: Applying MDE Tools at Runtime: Experiments upon Runtime Models. In: Models@run.time, pp. 25–36 (2010)
27. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model Checking Programs. Autom. Softw. Eng. 10(2), 203–232 (2003)
28. Warmer, J., Kleppe, A.: The Object Constraint Language: Precise Modeling with UML, 2nd edn. Addison-Wesley (2003)