Checking the Correspondence Between UML models and Implementation

Selim Ciraci, Somayeh Malakuti, Shmuel Katz and Mehmet Aksit

Outline

1. Introduction
   1. Current Approaches to runtime verification.
   2. Software design models: UML class and sequence diagrams.
   3. Using these models in RV and current approaches.
   4. Problem Statement
2. Our Approach
   1. Tier 1: Compilation.
      1. Simulation of UML sequence diagrams
      2. Observer generation.
   2. Tier 2: Execution
   3. Tier 3: After Execution, the conformance checking.
3. Example Application of the approach
4. Conclusion and Future Work

1.1 Current Approaches to RV

- Runtime verification usually involves:
  - A specification of the execution to be verified.
  - Runtime observers that monitor the execution events.
    - E.g. execution of methods.
  - Verifier: compares the monitored events with the specification.
- The specification is usually expressed with:
  - A state machine, a temporal logic formula.

1.2 Software design and execution process

- If we assume a software design process where UML is used for modeling various aspects of the software system before implementation:
  - The class structure is modeled with class diagrams.
  - The desired execution of the software is modeled with sequence diagrams.
  - These two diagrams contain the information needed to specify an execution.
    - After the software is implemented, these diagrams can be used in RV.

1.3 Using UML class and sequence diagrams in RV

- Can we use UML class and sequence diagrams as specifications in RV?
  - Our answer is YES.
- The benefits of doing so:
  1. UML is widely used and, hence, it provides a more familiar environment to the developers for using RV.
  2. The executions are already specified in sequence diagrams. There is no need to specify them again.
  3. It allows the conformance between UML sequence diagrams and the implementation to be checked.

Approaches to using UML in RV

- Approaches to consistency checking between sequence diagrams and the actual execution:
  - Aspects are used to observe (log) the execution of the software system.
  - The sequence diagram is *traced* with the observed execution.
- We observed the following drawback: Tracing is realized over a single sequence diagram.

1.4 Problem Statement

- Every possible use of the software system is usually modeled for mission critical systems.
  - Using a different RV for each sequence diagram can take too long.
  - Executions spanning over multiple diagrams should be considered.
  - So, we need to *combine* the sequence diagrams and *generate* a specification which is verified with RV.

Problem Statement

- When the desired executions of the software system is thoroughly modeled, how can we use these diagrams as specifications for RV?

2. Our Approach

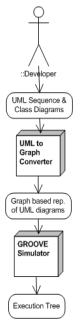
- The UML diagrams are combined by simulating the calls in the UML diagrams.
  - The simulation generates the *execution tree:* each path in the tree is an execution sequences of methods that is possible with the input diagrams.
- Using the sequence diagrams, the execution observers are generated.
  - These observers log the executed methods at runtime. This process *generates* the specification to be verified on the UML sequence diagrams.
- The execution tree is traced with logs for the verification.
- If a difference is located during trace, this can mean:
  1. Either it is a mistake in implementation
  2. Or, a new scenario not covered by the present set, and that should be added to the sequence diagrams

2.1 Overview

- Consists of three tiers:
  - Compilation: generation of runtime observers and the simulation of sequence diagrams.
  - Execution: Generation of the execution log.
  - After execution: Conformance checking between the logs and the sequence diagrams.

2.2 First Tier – Simulation of the Sequence Diagrams

- The simulation is realized through execution semantics modeled with graph transformations.
- Before simulation UML sequence and class diagrams are converted to graphs.
  - This step is automated.
- Simulation starts from a activator method.
  - Each invocation of the activator method is modeled in the sequence diagrams with asynchronous calls.
- GROOVE graph simulator is used for simulating and generating the execution tree.



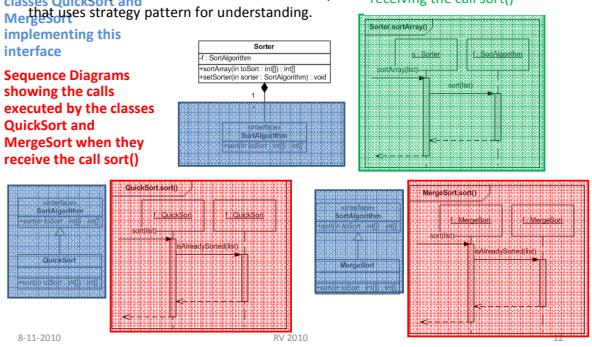
Simulation of Sequence Diagrams - Example

The interface SortAlgorithm and two classes QuickSort and MergeSort implementing this interface

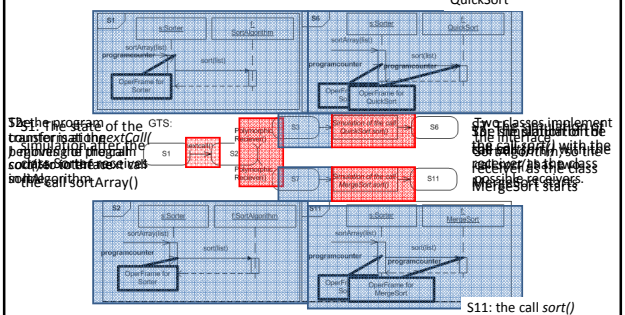
Sequence Diagrams showing the calls executed by the classes QuickSort and MergeSort when they receive the call sort()

We examplify the simulation over a small example that uses strategy pattern for understanding.

The sequence diagram showing the interface SortAlgorithm receiving the call sort()



Simulation of Sequence Diagrams - Example

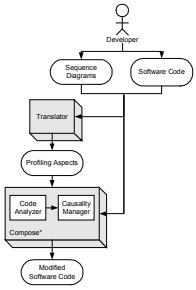


Simulation of Sequence Diagrams – Extracting information about executed methods

- For verification we need to learn names of the methods that have executed. We designed transformations that add the following information to the execution-tree:
  1. *executeMethod(activationCount, ClassifierName, ClassName, MethodName)*
  2. *returnMethod(activationCount, ClassifierName, ClassName, MethodName)*
- *ActivationCount:* a counter that is incremented by each activation of the activator method.
- *ClassifierName:* is used for distinguishing between different instances of a class.
- *ClassName, MethodName:* is used for identifying the method that begin/end executing

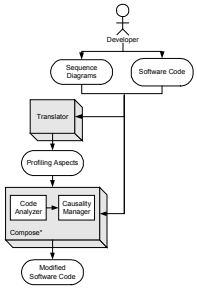
First Tier: Compilation Generation of the runtime observers

- We programmed a *Translator* that generates the runtime observers:
  - checks the static structure of the software to extract a list of methods defined in the code.
  - extracts information about the so-called activator method from each sequence diagram.
- The output of the translator is the *Profiling* aspect for each specified sequence diagram:
  - this aspect is generated in the language Compose*.
- The profiling aspect log information about the methods that are invoked during the execution of an activator method.
  - The *Profiling* aspects distinguish among multiple invocations of activator method by associating a unique identifier, called *ActivationID*, to each of them.
  - Consequently, separate log files are generated for each invocation of the activator method.



First Tier: Compilation Generation of the runtime observers

- The *Profiling* aspects are input to the Compose* compiler which generates the executable codes for the aspects and inserts them in the software code.
- In previous studies [1], the *Code Analyzer* is added to the Compose* compiler for detecting the inter-process communications within the context of an activator method.
  - If there is such a communication, the module *Causality Manager* modifies the invocation in both caller and callee sides with one more parameter holding *ActivationID*.



[1] Somayeh Malakuti, Christoph Bockisch, Mehmet Aksit: Applying the Composition Filter Model for Runtime Verification of Multiple-Language Software. ISSRE 2009: 31-40

2.2 Second Tier: Execution

- In the second tier, the runtime observers log the execution of the software system.
- The output of the observer is a set of sentences formed with the following words:
  - executeMethod(activationCount, ObjectID, ClassName, MethodName)
  - returnMethof(activationCount, ObjectID, ClassName, MethodName)
- *ObjectID:* is the runtime identifier of the object.

An Example Output From the Observers

executeMethod(1, 1807500377,sortArray,Sorter), executeMethod(1, 191550422 , sort, QuickSort), executeMethod(1, 191550422, checkSorted, QuickSort), returnMethod(1, 191550422, checkSort, QuickSort), returnMethod(1, 191550422, sort, QuickSort), returnMethod(1, 1807500377,sortArray,Sorter)

Method Sorter.sortArray() starts executing
Method QuickSort.sort () starts executing
Method QuickSort.checkSortert() starts executing
Method QuickSort.checkSortert() returns
Method QuickSort.sort() returns
Method Sorter.sortArray() returns

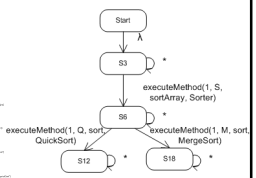
2.3 Third Tier: After Execution

- At this tier, the conformance between UML sequence diagrams and the execution is checked.
- The execution-tree is also a state machine.
  - If the sentence logged from an execution is accepted by the execution-tree, the logged execution conforms with the sequence diagrams.
- The output of the logger is sent to the model checker.
  - We use offline checking, though this check can be made online.

The Execution Tree

The Abstract Execution Automata

The conformance checker converts the state-space to a non-deterministic automata called *abstract execution automata*.
1.Remove intermediate transitions.
2.Add wildcard transitions.
3.Connect each activation with λ-transition



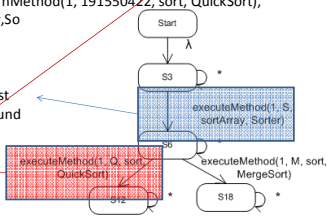
Binding the Object Ids

The *objectIDs* and the *activatorCounts* are treated as parameters. These parameters are bound to the actual values from the logged execution.

executeMethod(1, 1807500377,sortArray,Sorter), executeMethod(1, 191550422 , sort, QuickSort), executeMethod(1, 191550422, checkSorted, QuickSort), returnMethod(1, 191550422, checkSort, QuickSort), returnMethod(1, 191550422, sort, QuickSort), returnMethod(1, 1807500377,sortArray,So

Sorter.sortArray() seen first time -> 1807500377 is bound to S

QuickSort.sort () seen first time -> Q is bound to 191550422



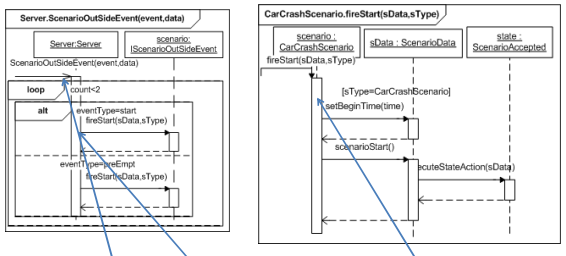
4. Application of the Approach

- We applied the approach to a crisis management system (CMS).
  - The original requirements descrpition is published in: *Jörg Kienzle, Nicolas Guelfi, Sadaf Mustafiz*, "Crisis Management Systems: A Case Study for Aspect-Oriented Modeling." TAOSD 7.
- CMS is used to manage the resources to treat a crisis.
  - Mission critical: errors can cause major problems.
  - Supports evolution: the CMS should be extendable with managers for different crisis types.
  - Prioritization: a crisis manager may request another crisis manager with a lower priority to release the allocated resources.

Application of the Approach

- To support these requirements the CMS is designed with events:
  - The operators (users) fire events which are handled by Crisis Managers -> programs that coordinate reporting and dispatching the police/ambulance.
  - Some important events: start of crisis, and start of allocation of resources to a crisis.
  - The crisis managers are prioritized: a crisis manager may request another crisis manager with a lower priority to release the allocated resources.

Design of CMS



Class Server is the event interface. Here the start of an crisis is fired

Class Server notifying the crisis managers

Car crash crisis manager handles the start of the event by allocating resources

Application of the Approach

- Prioritization is not used in the first version of the CMS.
  - First version is deployed with car crash crisis managers (implemented with class *CarCrashScenario*).
- Evolution: a crisis manager to handle presidential accidents needs to be added.
  - Has a higher priority then car crash crisis manager.
- The sequence diagrams show the prioritization event handled correctly by car crash crisis manager.
  - But how can we be sure the implementation also handles it?

Application of the Approach

- For the case study, we used the design of CMS with two crisis managers: PresidentialEmergency and CarCarshScenario.
  - 6 sequence diagrams: in total 36 actions, 4 conditional frames
  - We simulated a case where two accidents happen; the events start of crisis and start of resource allocate fired twice.
- The PresidentialEmergency has a higher priority then CarCrashScenario.
  - The sequence diagrams show that CarCrashScenario releasing the resource it allocates. But this sequence diagram is not correctly implemented.
  - Our aim is to catch this error.

Application of the Approach

- The simulation generated 20075 states and 21007 transitions.
  - The simulation starts with the Server receiving the events and making polymorphic calls to fire the events.
  - Took 2 minutes and used 37Mb of memory.
  - With the events executed twice, the state-space contains the following executions:
    - CarCrashScenario, PresidentialEmergencyScenario
    - PresidentialEmergencyScenario, CarCrashScenario
    - CarCrashScenario, CarCrashScenario
    - PresidentialEmergencyScenario, PresidentialEmergencyScenario
    - These sequences have branches where the actions within the fragments of the conditional frames are simulated.

Application of the Approach

- The detection of the inconsistency took 25 seconds.
  - Including the time to convert the state-space to the abstract execution automata.

5. Conclusions and Future work

- The approach looks promising
  - Sequence diagrams can be used as a specification for runtime verification.
  - The simulation shows implicit execution sequences which may be missed by developers.
    - Our approach can be used to capture errors in these sequences.
  - Either offline/online checking can be used.
- Possible directions on continuing this work:
  - Online checking/monitoring.
  - Add recovery when inconsistency is detected.
  - Language independent monitors.