Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG- 2008-IC-005

Jasmine: A Tool for Model-Driven Runtime Verification with UML Behavioral Models

Zhou Zhou, Linzhang Wang, Zhanqi Cui, Xin Chen, Jianhua Zhao

Postprint Version. Originally Published in: Proceedings of 11th International Symposium of High Assurance System Engineering (HASE08), IEEE Computer Society Press, 2008, pp.487-490.

Most of the papers available from this document appear in print, and the corresponding copyright is held by the publisher. While the papers can be used for personal use, redistribution or reprinting for commercial purposes is prohibited.

Jasmine: A Tool for Model-Driven Runtime Verification with UML Behavioral Models

Zhou Zhou, Linzhang Wang, Zhanqi Cui, Xin Chen, Jianhua Zhao
State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, P.R.China 210093
Department of Computer Science and Technology, Nanjing University, Nanjing, P.R.China 210093
lzwang@nju.edu.cn

For Research Only

***Abstract*— This paper describes the *Jasmine* tool to detect inconsistencies between the modelled behavior depicted in UML behavior models and monitored runtime behavior of Java programs by runtime verification. *Jasmine* takes Java programs under verification and corresponding UML models including sequence diagrams, activity diagrams and state machine diagrams. *Jasmine* imports and parses UML models created by any UML modelling tools. *Jasmine* instruments the code directed by the UML models, so as to relate the monitored runtime behavior to the UML models. *Jasmine* drives the instrumented program by existing test suites to collect the program execution traces. *Jasmine* checks the consistencies between the collected program execution traces and the UML models. It is implemented in Eclipse framework, working as a stand-alone Java application as well as a plug-in in Eclipse platform. It is highly automated and has been evaluated on several case studies.**

I. Introduction

Model-driven engineering based on UML shifts software development from a code-centric activity to a model-centric activity gradually since the models can be used to alleviate software complexity in an abstract level[1], [2]. Design-level UML behavioral models, such as UML sequence diagrams, activity diagrams and state machine diagrams, are widely used to specify and represent various aspects of the expected behavior of the program in the designer's viewpoint, and are also used to drive the realization of implementation in model-driven engineering. UML sequence diagrams are employed to describe interactive behavior between objects for use case scenarios. State machine diagrams are constructed to represent the possible states and transitions in the life cycle of an object for an individual class. Also, activity diagrams are designed to represent the realization of class operations. The behavioral models under study are supposed to be consistent with the requirement specification. However, in current research and practice of model-driven engineering, the code is still implemented by the programmers based on the design model rather than automatically generated from the design model. The consistency between the code and the model is still need to be determined.

Testing and verification are two effective solutions to find the discrepancies of the program with respect to its specification. In this concern, the UML behavioral models are the rational oracle against which conformance testing and consistency verification of the corresponding code implementations can be done. UML models provide the primary information, only describe the essential aspects of the specification. They have already been used to generate test cases. But, only abstract test cases can be derived directly from the abstract models. In most cases, concrete test cases can only be created manually, or partially automated by employing random method. The combination of testing and verification is a promising approach, named runtime verification, which is a lightweight approach to program reliability in a way that conduct verification on previous testing result. Its basic idea is to gather information during program execution and use it to conclude properties about the program, either during testing or in operation.

This paper describes the *Jasmine* tool, which helps programmers or modelers in detecting and tracking inconsistencies between the UML behavior models and Java implementations by runtime verification in testing phase. The tool is fully automated and does not require manual intervene or assistance during the verification. It can be used to provide consistency feedback, program bugs resulting from the wrong implementation,and the imperfect behavior models constructed in reverse engineering for legacy systems.This paper summarizes the technique and presents the tool and its capabilities.

The paper is organized as follows. In next section, we introduce the underlying technique of Jasmine. Section 3 introduces the tool architecture and the whole verification process. Section 4 introduces the evaluation of Jasmine based on some experiments. The related works are discussed in Section 5. Section 6 concludes this paper and discusses the possible extensions in the future.

II. The Underlying Techniques

A model-driven runtime verification approach is proposed to check whether the investigated behavior of the program is implemented as expected. The theoretical background was fully described [9], [10], [11]. This approach bridges the different levels of abstraction between the model and code. The base-lined behavioral models in design level are directly reused as the oracle for runtime verification, and drive the verification process. A set of model coverage criteria is defined by extending the code-level coverage criteria, to assess the adequacy of the verification as well as to control the verification process. Instrumentation mechanism is used to monitor runtime behavior and record them in an analyzable format for postmortem analysis of runtime behavior. However, the specified behavior is described in UML behavioral model,

1530-2059/08 $25.00 © 2008 IEEE
DOI 10.1109/HASE.2008.62

IEEE computer society

Authorized licensed use limited to: Nanjing University. Downloaded on September 15, 2009 at 03:47 from IEEE Xplore. Restrictions apply.

and the monitored runtime behavior is in an execution trace. To make them comparable directly, we need to relate the monitored behavior with the UML models. In our research, a behavior unit modelled in UML sequence diagram, activity diagram, and state machine diagram is defined as an event caused by method calling or method execution. Semantically, the behavior modelled in above UML diagrams could be depicted as a temporal order of method-level interaction.

The idea of model-driven verification is just to show the conformance of the expected behavior represented in the models and the implemented behavior of the program. First, the UML models are imported and parsed to extract the syntactic model constructs which can be used to imply behavior semantically. Second, when the implementation became baseline, the program is instrumented using the UML models as a guide, so as to relate the runtime behavior, i.e., execution traces, with the modelled behavior. Then the instrumented code is driven to execute with the previous generated test inputs. The program execution traces are monitored and logged to a trace file. Last, the observed execution traces are compared with respect to the UML models. It is straightforward to determine whether the modelled behavior is implemented as expected.

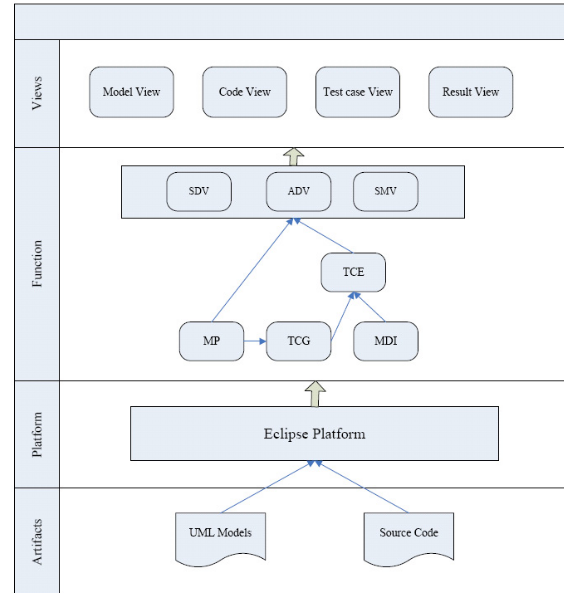

Fig. 1. The Tool Architecture

III. Tool Implementation

In the very beginning, we have developed a model-driven testing tool named UMLTGF with C++[9], which automatically generates abstract test cases from UML activity diagrams. This provides the groundwork for *Jasmine* tool suite implemented in Java. Since the executable concrete test cases could not be derived directly from the abstract UML design model, we moved on to test execution and runtime verification other than just focused on test generation. UMLTGF was extended by incorporating model-driven runtime verification with UML.

Jasmine was implemented in Eclipse framework, as a Java application first. And now, it is created in an Eclipse plug-in architecture, the interface of which are general components inherited from Eclipse platform. It can view the model, code, and test in same environment. Model-level test generation, code-level test execution and postmortem verification are integrated. The core function in the architecture of *Jasmine* is described in Figure 1. Figure 2 depicts a few screen snapshots of Jasmine, represents the verification result against UML sequence diagram,activity diagram, and state machine diagram, respectively. The tool could be downloaded in [7], including the executable version, demos and manual documents. The design and implementation of these components are detailed as follows.

A. UML Model Parser(MP)

The originally prototype UMLTGF was created as a plug-in integrated to the modelling tool, Rational Rose. UMLTGF can easily import and parse the UML specifications(called .MDL plain text file)with the help of Rose Extensibility Interface[3], then extract the modelled information and store them in the specific data structure so as to be accessed by the test case generator. Later on, we re-implemented the model parser with Java, using an open source project CrazyBean[4]. CrazyBean can only process the text format of UML produced by Rational Rose(.MDL file). Since all modelling tools provide translation functions to support model interchange by abiding by XMI standard, in order to make our tool vendor-independent, we re-implemented the model parser to support mete-data processing in *Jasmine*. **MP** imports and processes UML in XML format so as to make *Jasmine* work with any UML modelling tools(The version of UML depends on the modelling tools). By parsing the XML-based UML model, **MP** can extract behavior information from the design models, and re-establish the representation of the expected behavior, so as to provide facilities for further test generation from the models or verification runtime behavior against the models.

B. Model Directed Instrumentation(MDI)

For a Java program, **MDI** inserts some probe statements into its source code to investigate the method interaction, when the probe statements are executed, monitored behavior will be dumped to a trace file. The program execution traces gathered are a sequence of events corresponding to method interaction. Because we want to track the runtime behavior related to modelled behavior in the models, in order to reduce the cost of irrelevant information instrumentation, only the method interaction described in the models is focused on in **MDI**, and probes related to the model are inserted into the code. The insert position in the code is determined by the feature of the information to be collected. Traditional instrumentation is only concerned with the method execution. We want to track the runtime behavior in finer granularity. The probe statement of method calling is inserted before the

For Research Only

Authorized licensed use limited to: Nanjing University. Downloaded on September 15, 2009 at 03:47 from IEEE Xplore. Restrictions apply.

calling statement of method calling, and the probe statement of method execution is inserted before the first statement of the method definition of the method executing. Thus we can monitor method calling and method execution event pair at runtime execution. The dynamic behavior, such as method calling and method execution, can be profiled by the execution of the instrumented statement and recorded into a trace file. Currently, **MDI** can instrument Java source code, and Java Byte code with the help of BCEL(Byte Code Engineering Library)[6].

C. Data-Driven Program Execution

For a program under verification, we gather its execution traces by running its instrumented version. The program executions are driven by previous prepared test inputs in a data pool. During system-level program execution, all the test inputs are what the users need to provide by keyboard and/or mouse operation. We can also directly reuse the test data pool if there exists one for the system testing.

Due to absence of sufficient real-world data, creating suitably test data is often a difficult task. Random test generation is a black-box technique, and needs no information on the internal structure of a program other than the input type and domain. In this concern, randomness can increase the variety of input values so as to exercise and profile different behavior of a program under verification. Random method is also inexpensive charge and could be implemented in an automatic fashion.

In our approach, Jasmine provides a heuristic wizard, TCG, in interactive mode to customize the random test data generation. We assume that users have the knowledge of input type and domain. Users can specify the input sequence, type, domain, and sample number. This allows us to take advantage of randomness but still have control over test input generation for the program execution at the system level. Here we just handle simple input type such as integer, real, char, enumerable set, and so on.

Only the test inputs are not enough for execution of a program. How and when is the test data fed to the program? How the program is executed? These problems should be solved before execution. We create a driver in Jasmine, TCE, to ensure the program execution process in a mode without human intervention. TCE activates a program under verification, controls the execution, gets test data from the data pool, and feeds the test inputs to the program upon request. Currently, TCE follows Junit testing framework.

D. Model-Based Verification(MBV)

The most important concern of testing and verification is oracle. Traditionally, oracle is manually created based the requirement, or the verdict is done by engineers manually. *Jasmine* reuses the design models as the oracle of runtime verification. Any execution of Java programs is a sequence of method calling and method execution events at runtime. So the monitored behavior is a sequence of method call and method execution, which correspond to the message sends and

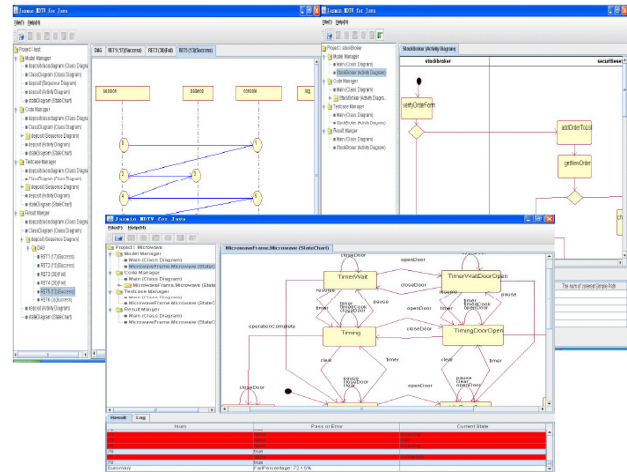

Fig. 2. The Panels of *Jasmine*

receives in the models because of the guided instrumentation. Thus, runtime execution trace can be compared directly with the UML models. Currently, *Jasmine* can verify the program behavior against UML sequence diagram, activity diagram and state machine diagram. Since above three diagrams model desirable different aspect of behavior with different notations and semantics, *Jasmine* provides a separate verification component for each diagram, such as SDV for sequence diagram, ADV for activity diagram, and SMV for state machine diagram in Figure 1. If a trace is not match any path in the UML model, *Jasmine* will report that an inconsistency is detected. we can conclude that some measures should be taken to drill into the corresponding test cases, program modules, and models, to locate and remove the inconsistency.

IV. Related Work and Evaluation

Testing tools only focus on observable system-level output during test execution. The verdict is made based on the comparison of observed output and expected output. Internal behavior is not considered by testing. Verification tools either employ model checking techniques with high formalism,or facilitate engineers with checklist for manual review. It is not easy to use formal verification techniques directly in industry because the specification languages in the verification tools are too formal and theoretical to master easily for engineers. Runtime verification makes use of the advantages of both testing and verification. The runtime verification techniques have been used for Java programs to monitor temporal properties and detect program errors such as deadlocks, data races, and memory leak. In those works, the specification languages are based on formal notations or event-based programming notations. Java MAC[12] , HAWK[13], JASS[14] define their own specification languages, which are based on temporal logic. Java-MOP[15], [16] is another runtime verification tool for Java programs. Its key feature is its extensible logic framework. In those works, the specifications need to be

For Research Only

Authorized licensed use limited to: Nanjing University. Downloaded on September 15, 2009 at 03:47 from IEEE Xplore. Restrictions apply.

elaborated based on programs, and it is difficult to reuse design or other specifications directly. Relatively, in our approach, a program under verification could be regarded as a black box, and the specifications in requirements and design could be directly reused.

Jasmine is a tool inherit advantages from both testing and verification tools. With *Jasmine*, we have conducted several case studies[9], [10], [11]. It did find some inconsistencies between the design models and implementations of these lab created projects, which result from the wrong implementation. From our experience in evaluation, the advantages of *Jasmine* are as follows. First, it significantly reduces the engineering effort in comparing an application's expected behavior and runtime behavior. Second, it needs no further formalism other than UML for the users. Third, instant feedback in coding phase of behavioral inconsistency between the models and implementation is a fundamental best practice in the software engineering. In this case, the defect could be removed as early as possible, thus reducing the cost of defect removal. Four, Jasmine employs instrumentation technique to monitor runtime behavior in addition to its traditional usage of collecting code coverage. Last, its open architecture makes integration and extension easy.

The field of runtime verification overlaps with the field of testing from the perspective of test oracles. Jasmine performs consistency checking off-line, which essentially leads a supporting tool for testing in which the UML-based specifications are used as automatic test oracles. The existing works on runtime verification have typically focused on program monitoring, which interleaves the analysis and recording program information with program execution. So far, *Jasmine* is not a commercial-grade product. In order to evaluate the scalability and usability of *Jasmine*, we try to find a appropriate real project in industry to apply our techniques. We have conducted an experiment with such a real project. This case study accidently indicates that *Jasmine* can help engineers verify whether their manually created design models are perfect with respect to the legacy system. So, in our opinion, how far *Jasmine* can go in the industry depends on the future of model-driven engineering.

V. Conclusion and Future Work

We have presented *Jasmine*, a tool implements the model-driven runtime verification approach. *Jasmine* integrates design specifications and implementations by rigorous verification for identifying inconsistencies between UML behavioral models and Java programs. It is tightly integrated with the most popular modelling and testing environment. *Jasmine* can be used to detect not only the program bugs resulting from the wrong implementation, but also the imperfect behavior models constructed in reverse engineering for legacy systems. To our knowledge, it is the first tool which generalizes runtime verification techniques to verify Java implementations against UML models. It can be applied directly to the real project designed with UML and implemented with Java in the industry since it only depends on UML and Java.

Using the Eclipse framework opens the potential to link *Jasmine* with a lot of other Eclipse plug-in contributions and aims to simplify the extension of the tool. *Jasmine* is an ongoing project. Currently, only UML sequence diagram, activity diagram and state machine diagram can be processed, and only consistencies related to the temporal sequence of method interaction can be identified in our tool suite. We plan to expand the approach to support other behavior models in UML 2.0 and other behavior categories, and to improve the scalability, stability and usability of *Jasmine* for applicable to real project. In addition, it is necessary to improve the consistency checking algorithms in Jasmine for producing the analysis result faster.

Acknowledgements

The authors are Supported by the National Science Foundation of China under Grant No. 60721002, 60603036, and by the National 863 High-Tech Programme of China under Grant No.2007AA010302, and by the Jiangsu Province Research Foundation under Grant No. BK2007139.

References

[1] Stuart Kent, Model Driven Engineering, Third International Conference on Integrated Formal Methods (IFM 2002), LNCS 2335, pp. 286-298, 2002.

[2] OMG, UML2.0 Superstructure Specification, available at http://www.uml.org, Oct. 2005.

[3] REI. ftp.software.ibm.com/software/rational/docs/v2002/Rose_REI_guide.pdf

[4] CrazyBean. http://crazybeans.sourceforge.net/

[5] Eclipse - an open development platform. http://www.eclipse.org/

[6] BCEL. http://jakarta.apache.org/bcel/index.html.

[7] Jasmine. http://cs.nju.edu.cn/lzwang/Jasmine/index.html.

[8] Mingsong Chen, Xiaokang Qiu, Wei Xu, Linzhang Wang, Jianhua Zhao, and Xuandong Li UML Activity Diagram-Based Automatic Test Case Generation For Java Programs The Computer Journal, Oxford Press, 2007, doi:10.1093/comjnl/bxm057.

[9] Wang, L., Yuan, J., Yu, X., Hu, J., Li, X. and Zheng, G., Generating Test Cases from UML Activity Diagram Based on Gray-Box Method. *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC2004)*, Busan, Korea, 30 Nov.- 4 Dec. 2004, 284–291, IEEE Computer Society, New Jersey.

[10] Li Xuandong, Wang Linzhang, Qiu Xiaokang, Lei Bin, Yuan Jiesong, Zhao Jianhua, Zheng Guoliang. Runtime Verification of Java Programs for Scenario-Based Specifications. In *Proceedings of the 11th International Conference on Reliable Software Technologies (AE2006)*, LNCS 4006, Springer, 2006, pp.94-106.

[11] Xuandong Li, Xiaokang Qiu, Linzhang Wang, Bin Lei, Eric Wong,UML State Machine Diagram Driven Runtime Verification of Java Programs for Message Interaction Consistency,the 23rd Annual ACM Symposium on Applied Computing, Vila Gal in Fortaleza, Cear, Brazil, March 16 - 20, 2008.

[12] M. Kim, S. Kannan, I. Lee, O. Sokolsky and M. Viswanathan, "Java-MaC: A Run-time Assurance Tool for Java Programs", In Electronic Notes in Theoretical Computer Science, Vol.55, Issue 2, Elsevier, 2001.

[13] M. d'Amorim, and K. Havelund, "Event-Based Runtime Verification of Java Programs", In Workshop on Dynamic Analysis (WODA 2005), 2005.

[14] D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim. "Jass- Java with Assertions", In Electronic Notes in Theoretical Computer Science, Vol.55, Issure 2, Elsevier, 2001.

[15] F. Chen, M. d'Amorim, and G. Rosu. "A Formal Monitoring-Based Framework for Software Development and Analysis", In Proceedings of ICFEM'04, volume 3308 of LNCS, pages 357-372, 2004.

[16] F. Chen, G. Rosu. "Java-MOP: A Monitoring Oriented Programming Environment for Java", In proceedings of TACAS'05, volume 3440 of LNCS, pages 546-550, 2005.

For Research Only

Authorized licensed use limited to: Nanjing University. Downloaded on September 15, 2009 at 03:47 from IEEE Xplore. Restrictions apply.