Software Applications Validation Environment: SAVE

● Sreeranga P. Rajan ● Indradeep Ghosh ● Oksana Tkachuk
● Mukul R. Prasad ● Praveen K. Murthy ● Ryusuke Masuoka
● Tadahiro Uehara ● Kazuki Munakata ● Kenji Oki ● Hirotaka Hara

(Manuscript received April 2, 2007)

The most vexing problem facing the software industry today is ensuring that complex heterogeneous client-server applications are defect-free. Fixing software defects in the field typically costs many times that prior to deployment. Customers becoming increasingly wary of security, privacy, and software safety may stop conducting business online. Current software validation techniques are largely inadequate. Conventional testing techniques are manually intensive, with unknown or poor functional (specification) coverage requiring the user to insert assertions in the source code. While formal verification techniques such as model checking offer 100% functional coverage, they cannot handle more than a few thousand lines of code. Members of Trusted Systems Innovation Group (TSIG) at Fujitsu Laboratories of America (FLA) have been working for more than a decade on developing novel techniques for model reduction, and the specification and validation of requirements. This paper describes software applications validation environment (SAVE), resulting from a close collaboration between TSIG and Software Innovation Laboratories (SIL) at Fujitsu Laboratories Ltd. SAVE weaves innovative techniques to provide an effective solution for validating the requirements for large heterogeneous software.

1. Introduction

Software failures are having a greater impact on consumers as technology rapidly becomes ubiquitous in affecting all aspects of our daily lives. Web applications are being deployed at an urgent pace to catch up with consumer demand for services online. The combination of software complexity and "speed-to-market" has put a tremendous strain on software quality. Software application providers have become vulnerable to losing business due to their resources being stretched thin in terms of software assurance. Conventional validation techniques centered on testing have become prohibitively expensive or simply unable to detect security and safety problems lurking deep within the code. Formal verification techniques developed in academia and industrial research laboratories have not proven effective beyond a few thousand lines of code.

There are two main reasons why current software validation techniques are inadequate in addressing the growing problem of software quality and assurance. A set of global requirements that a code base must satisfy can be obtained from the specification documents of the application being implemented by the code base. First, conventional testing entails inserting a number of assertions into the code base. It must be manually ensured that this set of assertions implies the set of global requirements to be satisfied by implementation. This can be an error-prone process. Moreover, generating test suites for individual modules comprising the

implementation, checking for assertion failures, and conducting diagnosis have become increasingly expensive and error prone along with a rapidly growing software code base of heterogeneous programming languages. Although the structural coverage of test suites can be calculated, the coverage of global requirements cannot be precisely computed. Secondly, advanced formal verification techniques such as model checking have two vexing problems: the inability to handle a few thousand lines of Java code, and the need for specialized arcane languages for stating requirements.

Software applications validation environment (SAVE) was developed to address both of the above problems that plague software validation. With SAVE, global requirements can be validated without having to manually insert assertions in the source code and writing expensive test suites. Secondly, SAVE employs powerful model generation and reduction techniques, as well as a user-friendly requirements specification language to facilitate model checking. These model generation and reduction techniques have enabled us to use SAVE on large software code bases consisting of a million lines of Java code.

2. Background

SAVE is the result of research conducted at the Trusted Systems Innovation Group (TSIG) of Fujitsu Laboratories of America (FLA) in collaboration with Software Innovation Laboratories of Fujitsu Laboratories Ltd., to address the validation of large heterogeneous software code bases. The primary goal of this work is to provide a framework that helps detect shallow and deep defects with a high degree of automation and usability prior to application deployment in the field. SAVE is being actively applied to validate large commercial Web applications. This paper focuses on the architecture of SAVE and the innovative techniques that provide the foundational basis for the various steps taken within SAVE. It also cites several Web application case studies to illustrate SAVE application.

Figure 1 shows a typical Web application consisting of three tiers. These layers embody different functional aspects of the application, are typically implemented in many different languages, and also distributed as “open” client-server applications. To deal with this complexity, SAVE provides the following three stages for automated validation of Web applications:

1) Environment generation

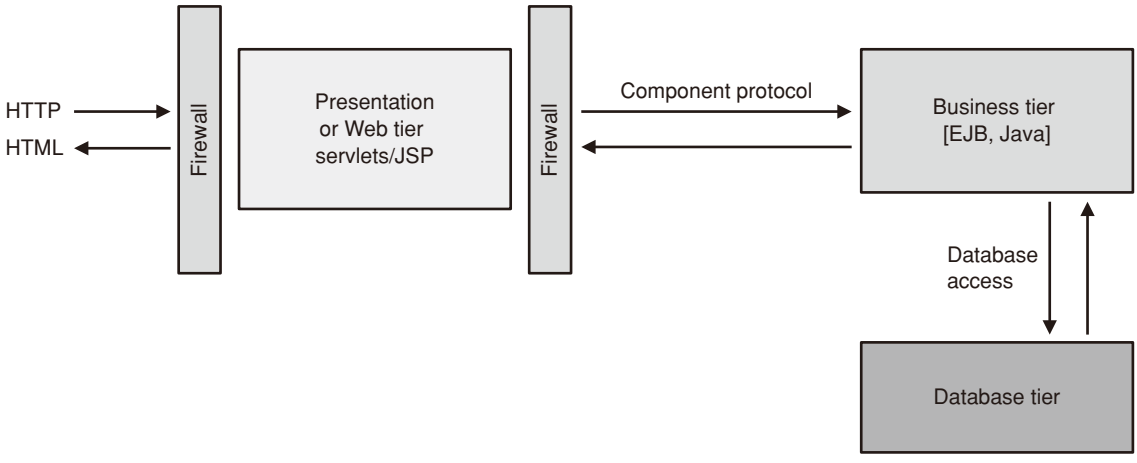

Figure 1
Three-tier Web application architecture.

2) Requirements categorization and automatic monitor generation
3) Model checking

Environment generation is the first step in preparing a software application for automatic validation. The environment interacting with the software application is usually too large for validation or may be unknown due to the "open" or reactive nature of a distributed client-server application. Using environment generation, a small set of behaviors representing a subset of typical and atypical scenarios is generated and integrated to "close" the software module. This "closed" software module is then ready to be run as a stand-alone application for the use of validation techniques.

In the second step of requirements categorization and automatic monitor generation, depending on the nature of the Web application and set of requirements to be validated, requirements are matched with domain-specific templates and instantiated. The domain-specific templates are created as a priori. Every domain-specific requirement template has a corresponding monitor for checking the validity of the requirement when model checking is performed. These generic monitors are instantiated with specific program objects and events leading to a set of monitors particular to the software application being validated.

In the third step, model checking analyzes the state transition system corresponding to the software implementation. It detects whether a requirement stated as a property using a mathematical expression is true in a given state. If a property violation is found, the violating trace (called a counterexample) is recorded and presented to the user for inspection.

There are many aspects of a Web application, in particular the presentation layer or Web tier that cannot be validated by model checking alone. In this case, we provide a novel method of automatically generating test cases that guarantee 100% coverage with respect to the Web tier requirements.

3. SAVE: Architecture and tool flow

Figure 2 shows the architecture and process of performing validation in SAVE as explained in the following subsections. By using SAVE we can rapidly uncover bugs hidden in the Web tier, business control logic, business data flow, and concurrent database access routines.

3.1 Environment generator

Environment generation[1)] is a technique used in modular approaches[2)] that restricts analysis to a selected part of a program (called a module), while representing the module's context of execution (called environment) at a higher level of abstraction. The environment has two aspects: drivers that hold a thread of control, and stubs that do not. Given a module as a collection of Java classes, the environment generation techniques first automatically discover the interface between the module and its environment, and then generate code for drivers and stubs. **Figure 3** shows a common scenario where drivers make calls to the module, which in turn calls the stubs. In general, references between the module and its environment may be arbitrary; that is, stubs may have callbacks.

The environment generation tools support the modeling of various interactions between a module and its environment. In the domain of Web applications, the Java part of a given Web application, excluding libraries, is treated as a module. The drivers are modeled to reflect actions of a user interacting with the application through a browser. Libraries and non-Java artifacts are modeled as stubs. Regular expressions are used to describe common user scenarios for driver generation, while static analysis and domain-specific knowledge (such as deployment descriptor files) are used for stub development. Upon being generated, drivers and stubs are combined with the original application code to

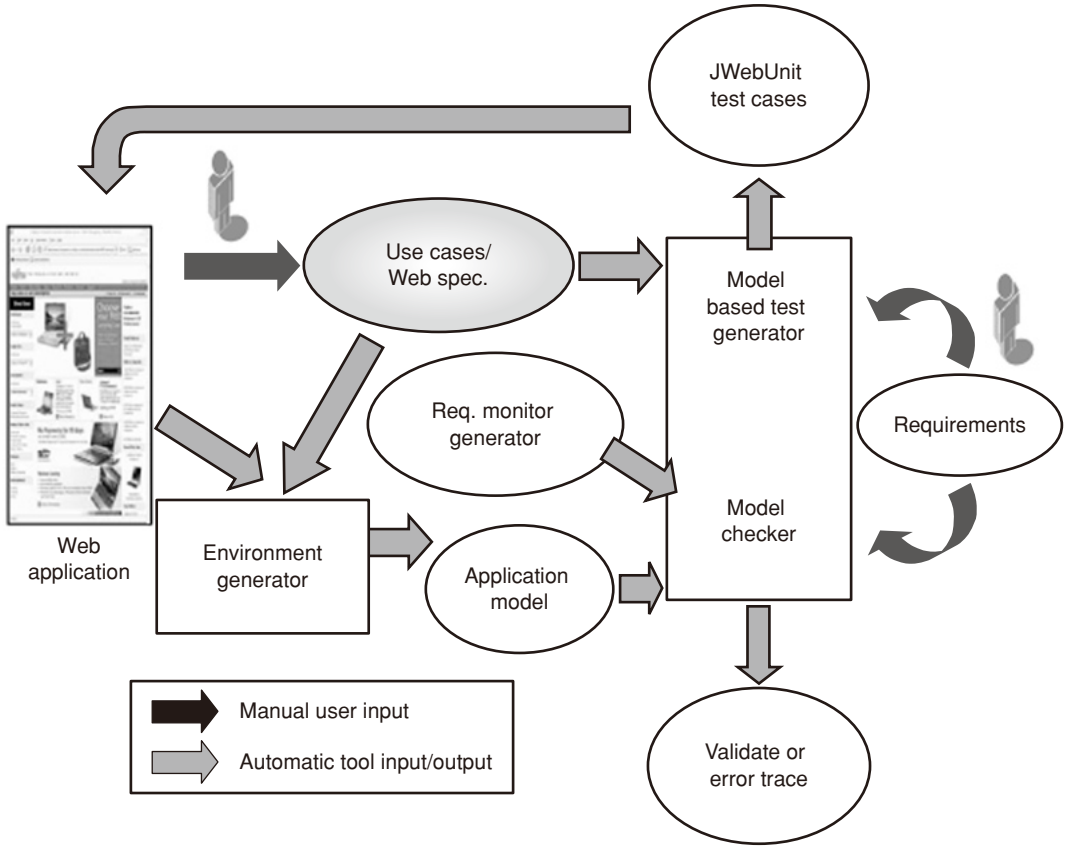

Figure 2
SAVE architecture and tool flow.

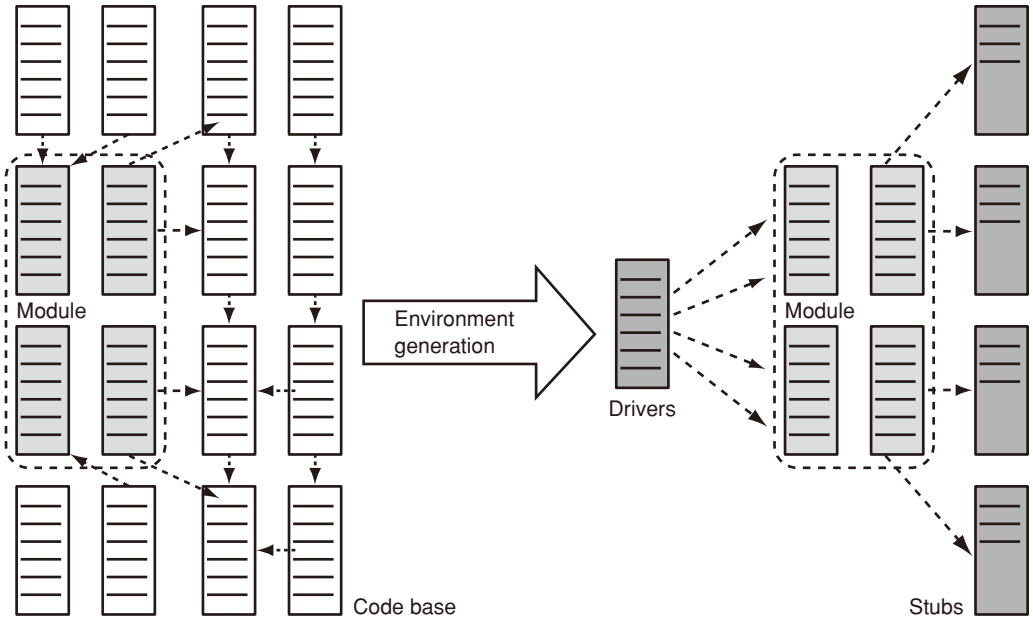

Figure 3
Environment generation.

create a self-executable Java program, ready for model checking.

3.2 Requirements monitor generator

The SAVE framework provides the user with a library of parameterized properties called requirements categories. This library is designed to encompass most requirements that would typically need to be checked in the context of E-commerce applications. The idea of restricting requirement specifications to a library of commonly used temporal logic formulas made available to the user was proposed in[3] among other works. This concept is developed further in several respects within SAVE and its application customized for E-commerce applications. In order to model check a given requirement, the user simply chooses a property template that models said requirement from the template library, as shown on the SAVE screen shot in **Figure 4**, and then provides the parameters necessary to specialize the template to the given requirement.

The SAVE framework automatically instantiates a monitor implementing this property and uses a third-party formal model checker to model-check it. For example, in order to model check the requirement "the shopping cart must be emptied after checkout in every shopping session," the user would simply choose the response property template, "***b*** follows ***a***" and supply the events **cart is emptied** and **checkout** for parameters ***b*** and ***a***, respectively. The SAVE framework automatically performs instantiation of the monitor for a specific property and subsequent model checking. In fact, even the task of specifying parameters for templates is considerably eased by providing the user with a library of principal events for the application at hand, as shown on the SAVE screen shot in **Figure 5**. The user has the option of choosing from among these events or supplying an original one. SAVE generates the library of events through static analysis of the application source code during the environment generation phase.

3.2.1 Model checker

The model checker we use in SAVE is called Java Pathfinder (JPF).[4] In JPF, requirements can be specified as assertions embedded in the code or as global monitors (called *listeners* in JPF terminology) that the user must create for each property by using the listener framework provid-

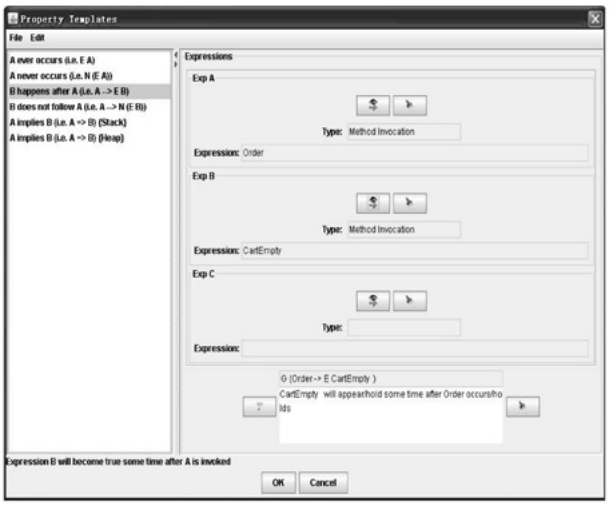

Figure 4
Requirements categories in SAVE.

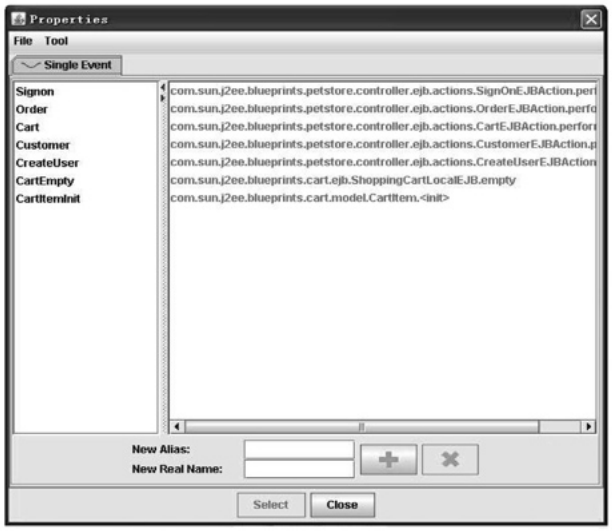

Figure 5
Universal event list and mapping to code.

ed with JPF. JPF was originally developed at NASA and has now been released into the public domain. JPF is an explicit state model checker built on top of a customized virtual machine that can run a Java program along all possible paths, checking for runtime errors, deadlocks, and race conditions. Though model checking is a powerful technique, there are two major complications that arise when model checking Java Web applications. First, the state transition system must be self-executable, ready to run on a single JVM, and written in pure Java. Web applications, on the other hand, are open distributed systems usually comprised of artifacts written in many languages (e.g., Java, JavaScript, HTML, XML). Secondly, for infinite domains, the state transition system for real software is infinite. In order to be tractable, model checking must be combined with powerful reduction techniques such as partial order reduction,[5),6)] data abstraction,[7)] slicing,[8)] and modular approaches.

3.2.2 Symbolic execution

Symbolic execution is a powerful model checking technique built into SAVE. In this technique the inputs are symbolic instead of being concrete values. A symbolic decision procedure engine is then used to check whether a certain requirement is satisfied.[9)] Symbolic execution provides precise path analysis and characterizations of all possible executions up to a certain bound. It is better suited to programs involving many arithmetic operations and properties, and can provide complete coverage of the system on which it runs. As the number of conditionals increase across long paths in a program, however, the path conditions can become too complicated to be eventually solved by the decision procedure in a reasonable period of time. Hence, environment generation should be used to also localize the application of symbolic execution.

3.3 Automatic model-based test case generation

In applying environment generation followed by model checking to E-commerce applications, the Web and database tiers are typically substituted with drivers and stubs, respectively. Thus, requirements directly related to the Web tier (composed of HTML/Javascript/JSP) or database tier (composed of JDBC/database) cannot be validated. In order to address this issue, we have developed a novel technique for automatically generating test cases corresponding to requirements related to these tiers.

It is clearly evident that constructing effective tests is no trivial matter. Existing work on testing Web applications tends to require testers who have expert knowledge about low-level details of the implementation. In addition, propositional abstraction — using propositions in abstracting an application — is still commonly used.

Instead, we use the specification model of a Web application written in a language called WAVE[10),11)] developed by the database group at UC San Diego to describe how screen transitions occur and how such transitions are linked to the database. We apply model checking to the WAVE specification based on the requirements, and automatically generate JWebUnit test cases corresponding to the requirements. These test cases can be run directly on the original application. **Figure 6** shows examples of a WAVE specification, requirement, and the corresponding JWebUnit test case generated.

Before describing our approach for testing Web applications, we first introduce the notion of Web test cases. A Web test case is a specialized program that performs user inputs and navigation on actual Web sites, and makes assertions along the way. A Web test case is considered to pass when it represents valid execution of the Web site, with all assertions being true. Otherwise, it is considered to fail. In our experiments, Web test cases are in the form of Java

programs using JWebUnit libraries.

In briefly describing our verification-based testing approach, we first note that the implementer of the application is likely to be other than the tester, who now produces a high-level specification model of the application and desirable properties to be verified. This immediately presents a couple of potential problems:

1) There is no guarantee that the specification model (written by the tester) is faithful with respect to implementation (written by the implementer).
2) There might be flaws in the specification of the Web application.

In addition to the original problem,

3) There might be errors in the implementation.

In this approach, we develop the specification and refine it. The specification is developed in a variety of ways. The tester can examine the implementation and manually create WAVE models. We could also create models in other similar formalisms like Scenery[12] (which uses hierarchical message sequence charts)[13] as shown in **Figure 7**, or by using unified modeling language (UML).[14] Automated code analysis tools can be used to reverse engineer the code base to produce UML-type models. Server logs and network traffic analysis can be used to construct scenarios and use cases of how Web pages are typically traversed.[15] Once the specification model has thus been developed, a property is verified through model checking of the specification model using standard techniques and the counterexamples obtained (until none exists), and then mapped to a Web test case. The Web test case is executed on the implementation. If it fails, problem 1) above is identified; if it passes, problem 2) above is identified. In either

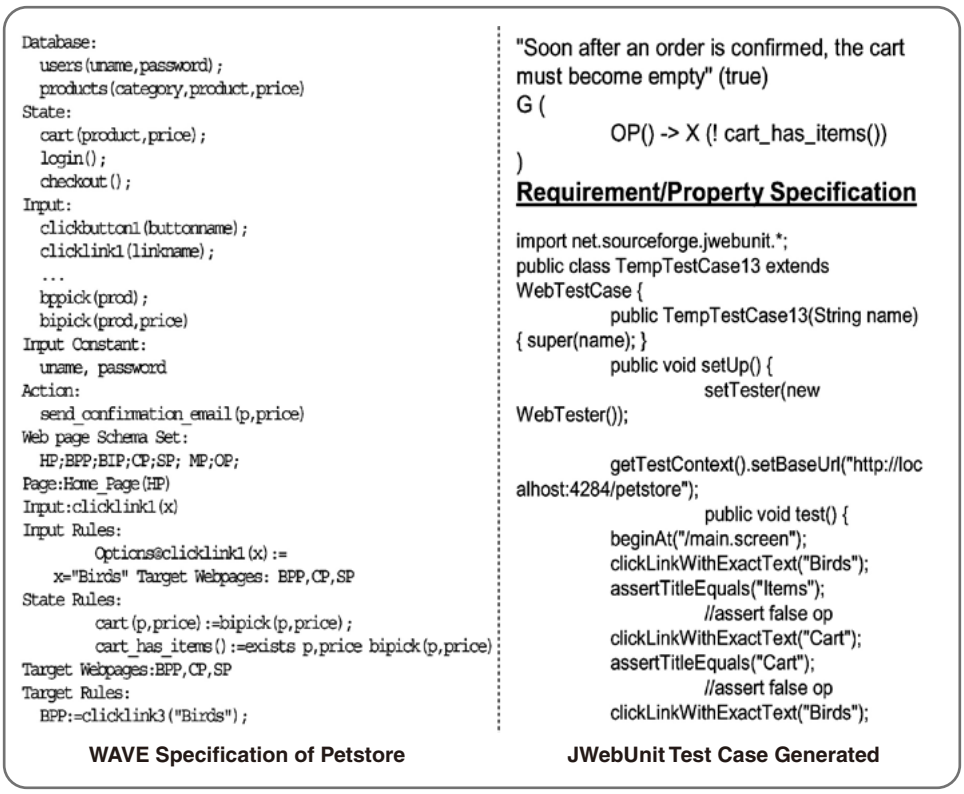

Figure 6
Examples of WAVE specification, requirement/property specification, and generated JWebUnit test case/monitor.

case, the tester properly modifies the specification as necessary. This process is repeated on all available properties and counterexamples. No previous work has successfully or credibly provided such a synergistic methodology. Our test generation methodology produces tests based on both user-defined properties (such as "shopping cart must be empty after checkout") and a comprehensive scenario analysis of the specification model. Assertions based on properties to be checked are automatically generated and inserted in test monitor code that implements the Web test case.

4. Case studies

We now present an external case study and the internal trials we conducted to evaluate the effectiveness of the SAVE tool. The first example is a public domain application bundled with the Java release. The second case study is a set of internal trials.

4.1 Java Pet Store

We applied SAVE to validate a widely used industrial Web E-commerce application called Java Pet Store.[16] We applied all three features available in SAVE to validate different types of requirements. Some examples of requirements are:

1) Shopping cart becomes empty after order confirmation: validated by environment generation and model checking.
2) If the order exceeds $500, then the status becomes pending: validated by symbolic execution.
3) It is not possible to check out if the cart is empty: validated by test case generation based on model checking.

Next, we explain finding defects in the Java Pet Store application. We applied our environment generation tools to the Java Pet Store application in the following way. The application code was treated as a module, calls to library methods (e.g., J2EE and JDBC libraries) were stubbed out, and the drivers were generated to simulate user interactions with the application (e.g., clicking on available buttons or entering information in a text box). If necessary, the tools are capable of automatically generating drivers that perform *all possible sequences* of button clicks and user inputs. But since such drivers are impractical, user specifications that describe likely interaction scenarios can be used to generate more practical drivers. The users only need

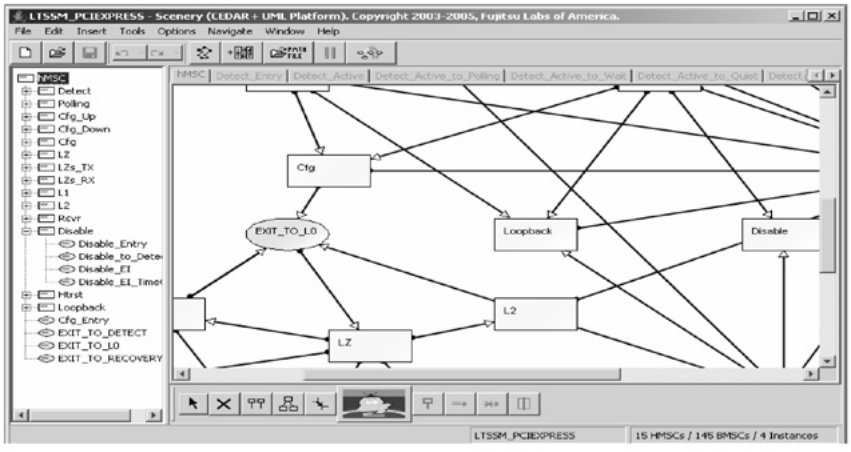

Figure 7
Scenery screen shot.

to specify the sequences of events performed. Our environment generation tools automatically generate appropriate event values such as user input data. **Figures 8** and **9** depict the structure of the original petstore application and its model after the environment generation step.

The stubbed Java Pet Store application, in combination with the generated driver, constitutes a model of the Java Pet Store application. This model is then model-checked. The generated Java Pet Store model was given to the JPF model checker and the global monitors were invoked. The model checker in SAVE reported a security violation of the requirement for the user to create an account with a second sign-in password matching the first. **Figures 10** and **11** show the output visualization for diagnosis. In addition, we found two other defects in the Web and database tiers using test case generation based on model checking: 1) a user can create an account with an empty profile and still proceed with E-commerce transactions, and 2) reusing a user name while creating a new user account crashes the system.

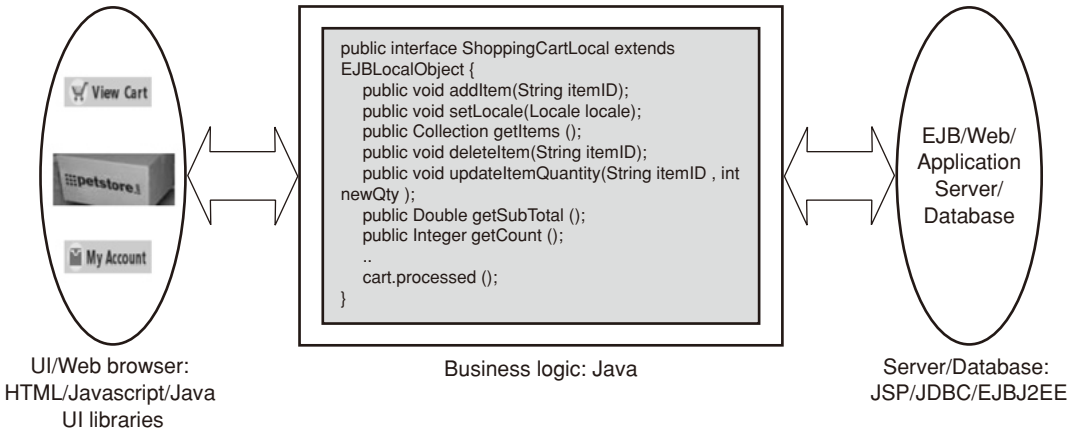

Figure 8
Original petstore application.

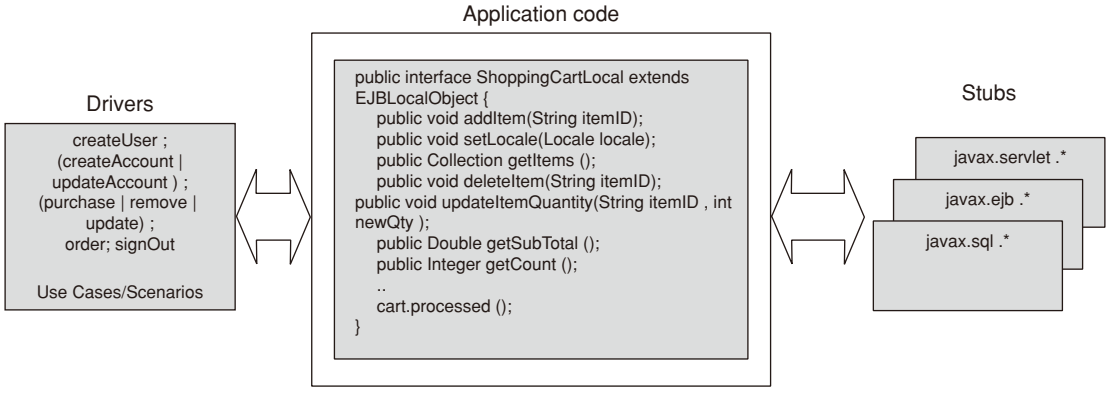

Figure 9
Petstore model.

4.2 Internal trials

SAVE has been utilized to detect defects in large software code bases written in Java for Web-based applications used by Fujitsu customers. The applications may consist of Java applets that communicate asynchronously with a server and various IO device controllers like printers and image readers. The applications are multi-threaded in nature and therefore very suitable for model checking to look for bugs arising from deadlocks and races that occur due to improper synchronization among different threads.

The first step in the model checking process is environment generation as shown in **Figure 12**. The application under verification is converted into a self-contained executable Java model. This conversion, currently semiautomatic in nature, is done using the techniques described earlier. The environment generator tool in SAVE can provide much assistance to the verification engineer in creating the drivers and stubs necessary, but might require some manual intervention to complete the process.

Following creation of the model, a bug was first introduced to verify whether JPF could catch the bug in the model. Initially this was not possible due to vastly increased state space in the model. This was tackled using three approaches. First, the system was initialized in a sequential manner to reduce any increase in state space in the uninteresting initialization phase. Once initialization was complete, the different concurrent threads were allowed to run in parallel. Although such modification is currently done manually, this technique may be incorporated automatically in the environment generation phase. Secondly, certain variables shared among different threads were marked as being unimportant from the standpoint of JPF and excluded from its analysis, by simply modifying the input properties file that holds

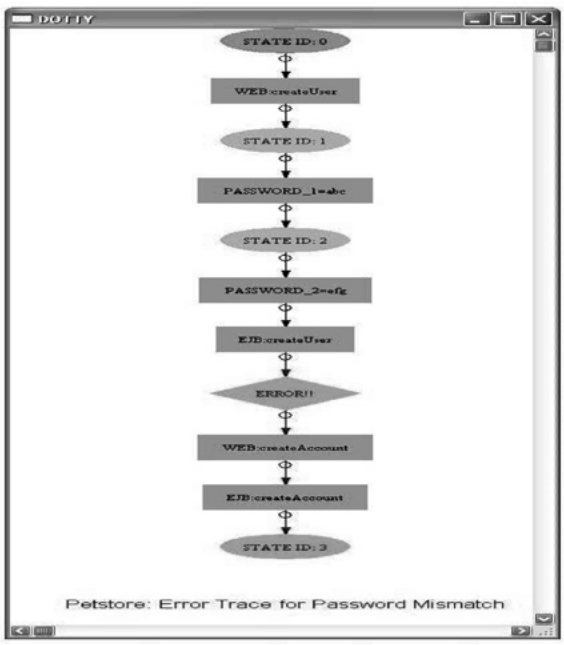

Figure 10
Output visualization of Error Trace.

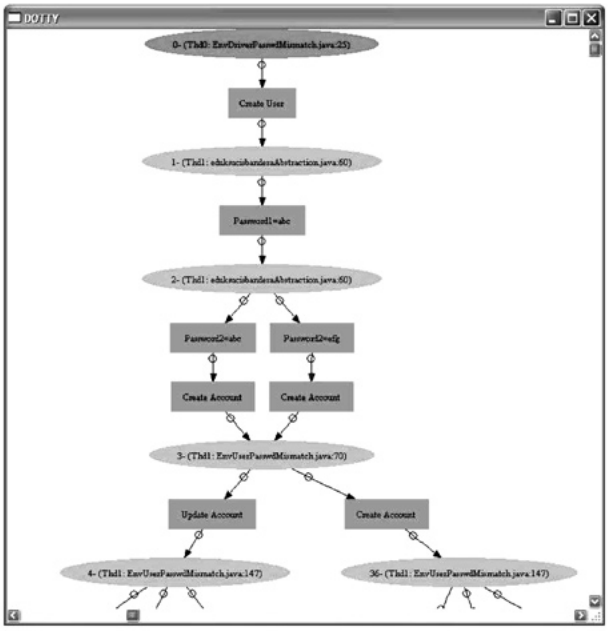

Figure 11
Visualization of all states and paths explored by model checking.

the parameters used by JPF during its execution run. This helped reduce the unnecessary branching that would occur if JPF tried to detect race conditions among these variables. This exclusion of variables must be done carefully since it may mask races that are actual bugs in the program. One approach is to exclude variables in the classes or methods that need not be verified at that time. Thirdly, a depth-first search was employed instead of a breadth-first search to keep memory requirements under control as only one path was checked and discarded at a time. Again this required minor modifications in the properties file of JPF.

It took about 3.5 hours of CPU time to complete the model checking of a typical Fujitsu Web application. There were some surprises during the detection of bugs in that paths to the defective states exposed by the model checker were unexpected scenarios. This illustrated that, unlike testing, model checking can indeed uncover unexpected program behavior. By using the technique above in addition to the injected error, JPF could also detect a deadlock situation in the application model.

5. Conclusions

SAVE is a powerful framework that encapsulates the most advanced techniques in software validation. With this framework one need not insert assertions in the application code base, but can simply check whether a set of global requirements is valid in a software application with a high degree of automation. If the validation fails, SAVE provides a visualization of the bug trace and can therefore help diagnose and trace the cause efficiently. We currently have about 25 requirements (properties) that Web applications must satisfy. These 25 requirements are generic across a variety of Web applications that include the aspects of navigation, business logic, security, and databases.

SAVE has been applied to several large open source and commercial applications, and successfully uncovered both previously known as well as new security and safety-critical defects. Some of these defects were hidden deeply in the execution of applications where it would have been almost impossible for conventional testing techniques to find. Our goal for the future is to make SAVE an indispensable tool for software development and testing teams seeking to avoid costly software failures in the field.

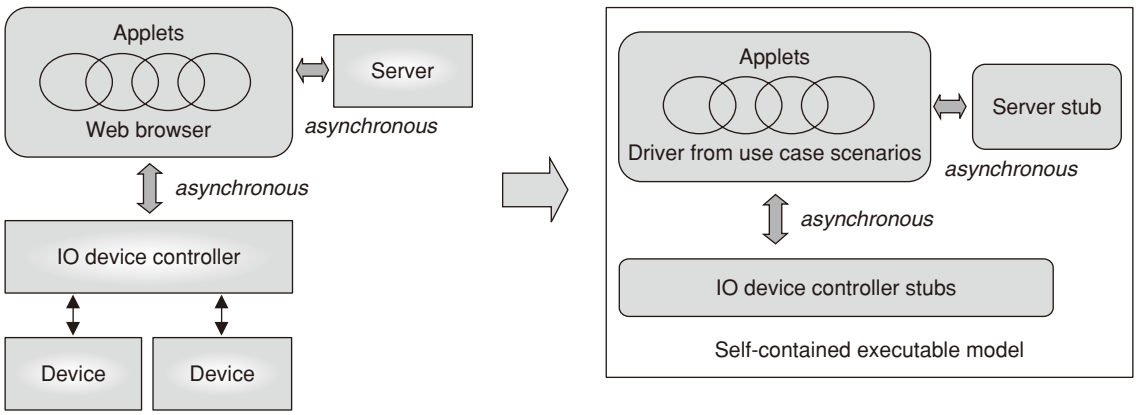

Figure 12
Environment generation for Web applications.

References

1) O. Tkachuk et al.: Application of Automated Environment Generation to Commercial Software. International Symposium on Software Testing & Analysis, 2006.
2) B. Hailpern et al.: Modular verification of concurrent programs. Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, Mexico, 1982, p.322-336.
3) M. B. Dwyer, G. S. Avrunin, and J. C. Corbett: Patterns in Property Specifications for Finite-state Verification. International Conference on Software Engineering, 1999.
4) W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda: Model Checking Programs. *Automated Software Engineering Journal*, **10**, 2, 2003.
5) G. Holzmann et al.: An Improvement in Formal Verification. FORTE 94, 1994.
6) P. Godefroid: Partial-Order Methods for the Verification of Concurrent Systems. Lecture Notes in Computer Science, 1032, Springer-Verlag, 1996.
7) C. Flanagan et al.: Predicate Abstraction for Software Verification. Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2002.
8) A Survey of Program Slicing Techniques. *Journal of Programming Languages*, **3**, 3, p.121-189 (1995).
9) C. S. Pasareanu et al.: Verification of Java Programs using Symbolic Execution and Invariant Generation. Proceedings of the 11th International SPIN Workshop on Model Checking of Software, 2004.
10) A. Deutsch et al.: Specification and Verification of Data-driven Web Services. PODS 2004.
11) A. Deutsch et al.: A system for specification and verification of interactive, data-driven web applications. SIGMOD Conference 2006.
12) P. K. Murthy et al.: High Level Hardware Validation using Hierarchical Message Sequence Charts. IEEE International High Level Design Validation and Test Workshop, 2004.
13) ITU Recommendation Z.120, Message Sequence Charts. Telecommunication Standardization Sector, Geneva, 1996.
14) OMG Unified Modeling Language Specification. Version 2.0, 2006.
15) P. Tonella et al.: Dynamic Model Extraction and Statistical Analysis of Web Applications. Proceedings of the Fourth International Workshop on Web Site Evolution, 2002.
16) Java Pet Store. *http://java.sun.com/j2ee/1.4/download.html#samples*

Sreeranga P. Rajan, *Fujitsu Laboratories of America, Inc.*
Dr. Rajan is a senior research staff member leading the software validation project conducted by the Trusted Systems Innovation Group (FLA). He joined FLA in 1996 from the Computer Science Laboratory at Stanford Research Institute International (SRI). Since then, he has worked on developing compilers and pioneering advanced software verification techniques at FLA. Dr. Rajan has numerous publications, patents, and awards. He currently serves as the founding Editor-in-Chief of ACM Transactions on Storage.

Indradeep Ghosh, *Fujitsu Laboratories of America, Inc.*
Dr. Ghosh received the Bachelor of Technology degree in Computer Science and Engineering from the Indian Institute of Technology, Kharagpur, India, in 1993, and the M.A. and Ph.D. degrees in Electrical Engineering from Princeton University, Princeton, New Jersey, in 1995 and 1998, respectively. In 1998 he joined Fujitsu Laboratories of America, Sunnyvale, California where he is currently a research staff member in the Trusted Systems Innovation Group. He has authored or co-authored more that 40 technical articles in international journals and conferences, and holds five US patents. He co-authored a paper that won the Honorable Mention Award at the International Conference on VLSI Design (1998). He has given numerous presentations at international conferences and workshops. At Fujitsu he has been involved in the design, verification, and testing of hardware and software systems. He is a member of ACM and senior member of the IEEE.

Oksana Tkachuk, *Fujitsu Laboratories of America, Inc.*
Ms. Tkachuk joined FLA in 2005 and works on environment generation techniques for commercial software. As part of her work at NASA Ames Research Center (during summer 2001, 2002, and 2003) and for her doctoral thesis (at Kansas State University in 2007), she developed automated environment generation techniques that support both synthesis from user specifications and extraction of environment models from implementations using static analysis techniques.

Mukul R. Prasad, *Fujitsu Laboratories of America, Inc.*
Dr. Prasad received the Bachelor of Technology degree in Electrical Engineering from the Indian Institute of Technology, Delhi, India, in 1995, and Ph.D. degree in Electrical Engineering and Computer Sciences from the University of California at Berkeley, California, in 2001. Since 2001 he has been with Fujitsu Laboratories of America in Sunnyvale, California where he is currently a research staff member in the Trusted Systems Innovation Group. He has authored more than 20 technical articles in international journals and conferences, holds three US patents, and won a Best Paper Award at the Design Automation & Test in Europe (DATE) Conference in 2002. His research encompasses all aspects of the validation of hardware and software systems. Dr. Prasad is a member of ACM and IEEE.

Praveen K. Murthy, *Fujitsu Laboratories of America, Inc.*
Dr. Murthy received the Bachelor of Science degree in Electrical Engineering from the Georgia Institute of Technology, Atlanta, in 1989, and M.S. and Ph.D. degrees in Electrical Engineering and Computer Science from the University of California at Berkeley, in 1993 and 1996, respectively. He then joined Cadence Design Systems, where he worked on dataflow compiler optimization algorithms. After Cadence, he joined in the startup of Angeles Design Systems, where he was lead architect of System Canvas (a system level design tool based on dataflow), as well as discrete-event simulators and an optimization engine for DSP and telecommunications systems. In 2001 he joined Fujitsu Laboratories of America, Sunnyvale, California where he is currently a research staff member in the Trusted Systems Innovation Group. He has authored two books and more than two-dozen technical articles in international journals and for conferences. At Fujitsu he headed the Scenery project and was involved in the verification and testing of hardware and software systems. He is a member of ACM and senior member of the IEEE.

Ryusuke Masuoka, *Fujitsu Laboratories of America, Inc.*
Mr. Masuoka is director of the Trusted Systems Innovation Group at Fujitsu Laboratories of America, Inc. at College Park, Maryland, USA. He is also an adjunct professor of UMIAC, University of Maryland, USA. Since joining Fujitsu Laboratories Ltd. in 1988, he has researched neural networks, simulated annealing, and agent systems. Results from all these research areas have led to products from Fujitsu. After joining Fujitsu Laboratories of America, Inc. in March 2001, he was engaged in research on pervasive/ubiquitous computing, Semantic Web, and bioinformatics, from which task computing is derived. He has now extended his research to trusted computing, software/security validation, and system level design.

Tadahiro Uehara, *Fujitsu Laboratories Ltd.*
Mr. Uehara received the B.E. degree in Control Engineering and M.E. degree in Intelligent Science from Tokyo Institute of Technology, Tokyo, Japan in 1993 and 1995, respectively. He joined Fujitsu Laboratories Ltd, Kanagawa, Japan in 1995, where he has been engaged in research on software engineering, especially object-oriented technologies and testing technologies for business applications.

Kazuki Munakata, *Fujitsu Laboratories Ltd.*
Mr. Munakata received the M.E. degree in Information Science from Japan Advanced Institute of Science and Technology, Ishikawa, Japan in 2001. In 2005 he joined Fujitsu Laboratories Ltd., Kanagawa, Japan where he has been engaged in research on software engineering, particularly software verification and requirement engineering.

Kenji Oki, *Fujitsu Laboratories Ltd.*
Mr. Oki received the B.E. degree in Information Science and Engineering, and M.E. degree in Computer Science from Tokyo Institute of Technology, Tokyo, Japan in 2004 and 2006, respectively. In 2006 he joined Fujitsu Laboratories Ltd., Kanagawa, Japan where he has been engaged in research on software engineering.

Hirotaka Hara, *Fujitsu Laboratories Ltd.*
Dr. Hara is the director of Software Innovation Laboratories (SIL) at Fujitsu Laboratories Ltd., Kawasaki, Japan. He received the B.S. degree in Information Science in 1984 and a Ph.D. in Information Engineering in 1992 from the University of Tokyo, Tokyo, Japan. He joined Fujitsu Laboratories Ltd., Kawasaki, Japan in 1984 and has been engaged in research and development of artificial intelligence and distributed enterprise systems. He is a member of the IEEE. He received the IPSJ Convention Award and the JSAI.