New Approach to Application Software Quality Verification

● Jun Ginbayashi ● Tadahiro Uehara ● Kazuki Munakata
● Kazuo Yabuta

(Manuscript received November 11, 2009)

Fujitsu has developed a technique that can automatically prepare test scenarios/data, run the tests, and check the results based on a formal description of the application's external specifications. This eliminates problems that can arise in conventional test methods, such as failure to consider all possible test cases or data values, the existence of too many test cases for exhaustive testing, and variation in test results due to personality-related differences. By combining this technique with an application framework, we have made it available for use in the development of full-scale applications. This paper introduces our work on using formal verification techniques to achieve ground-breaking improvements in software quality as part of the Production Innovation project in Fujitsu's SE division.

1. Introduction

Corporate business systems and social administration systems are becoming increasingly large-scale and complex. For example, the system of one financial institution has expanded from 5 million to 64 million steps since the introduction of the third-generation online system in the 1980s.[1] Meanwhile, system development timescales are becoming much shorter. For example, until 1998 the average development time was 11.6 months, but by 2002 it had fallen to 7.9 months.[2] Under these conditions, guaranteeing software quality is becoming a major issue. It has been reported that systems contain on average 122 bugs per million steps after entering service,[3] so there is strong demand for more sophisticated testing. However, with current testing techniques it is difficult to provide adequate quality guarantees for the following reasons:

1) Testing constitutes a major part of the overall development process (about 30% for ordinary business applications and over 50% for social administration systems or embedded systems).
2) Although various automation techniques have been proposed, it is difficult to get them accepted by the developer community.
3) As systems become more complex, it becomes harder for humans to elicit a sufficient range of test cases and prepare the necessary test data.

Formal verification techniques have attracted interest as a potential means of breaking through the limitations of conventional testing. Fujitsu has developed a technique for automating the creation of test scenarios and test data, running the tests, and checking the results based on a formal description of an application's external specifications.

In this paper, as part of the Production Innovation project at Fujitsu's systems engineers (SE) division, we focus on formal verification techniques, especially model checking techniques and introduce our efforts aimed at implementing these techniques in enterprise systems.

2. Formal verification techniques

Formal verification techniques are a class of so-called formal methods[note)] that mathematically guarantee the accuracy of software. In recent years, formal verification has attracted attention because of its potential to break through the current limitations of testing.[4)–6)] Organizations such as Japan's Ministry of Economy, Trade and Industry are also becoming increasingly aware of the importance of using formal methods to improve the reliability of future systems.[7)]

Formal verification is characterized by the software to be verified being represented as a model and then mathematically proven methods being used to investigate whether or not this model satisfies the properties to be checked. Instead of hunting for and eliminating bugs, this approach focuses on certain specific properties (e.g., absence of deadlocks, logging of all transactions, etc.) and guarantees that these properties are always satisfied. In actual software development processes, formal verification can be applied in two phases:

- Analysis/design phase: Verification of specifications
- Build/test phase: Verification of source code

When formal verification is applied to the analysis/design phase, there are expected to be benefits resulting from the ability to detect specification errors at an early stage, but this makes it necessary to express the specifications as a mathematical model, which is a very difficult task. On the other hand, its application to the build/test phase has the major benefit that the source code can be verified directly, which means there is no need to construct a mathematical model and the quality of the end product can be guaranteed.

Some typical examples of how formal verification has previously been applied are shown in **Figure 1**. Although it has been used to verify the specifications of some embedded systems and business systems, its application to source code verification has so far been biased towards embedded systems. In our work described below, we dealt with source code verification.

	Embedded systems	Business systems
Verification of specifications	• NASA Mars probe (SPIN) • FeliCa IC chip (VDM) • Photocopier firmware (SPIN) etc.	• Paris Metro control system (B-Method) • Mission-critical transaction management (Z) etc.
Verification of source code	• Windows device driver (SDV) • C source code etc.	• Business applications (Web applications developed in EZDeveloper)

Parentheses give the names of formal verification techniques.

Target of this technique

Figure 1
Example applications of formal verification techniques.

3. Model checking techniques

Among formal verification techniques, attention has been drawn to model checking techniques, which can perform verification relatively mechanically. A model checking technique treats software as a finite state transition model and automatically checks whether or not this state transition model satisfies the separately stated properties. An example of model checking based on an explicit state search is shown in **Figure 2**.

From a state transition model, all the possible paths (a→c→d→e ... etc.) are generated, and automatic checks are performed to see if the properties are established at each state. If a state is found to contravene a property, then one path that arrives at this illegal state (a→b→c′→e ...) is presented as a counterexample. Since this path can be used to reproduce the same error, it provides useful information for debugging.

Although the applications of model checking have so far been centered on embedded software, at Fujitsu Laboratories we have been investigating applications to the quality assurance of business applications since 2006. We have also been

note) Techniques for the specification, development, and verification of software and hardware based on logic and mathematics.

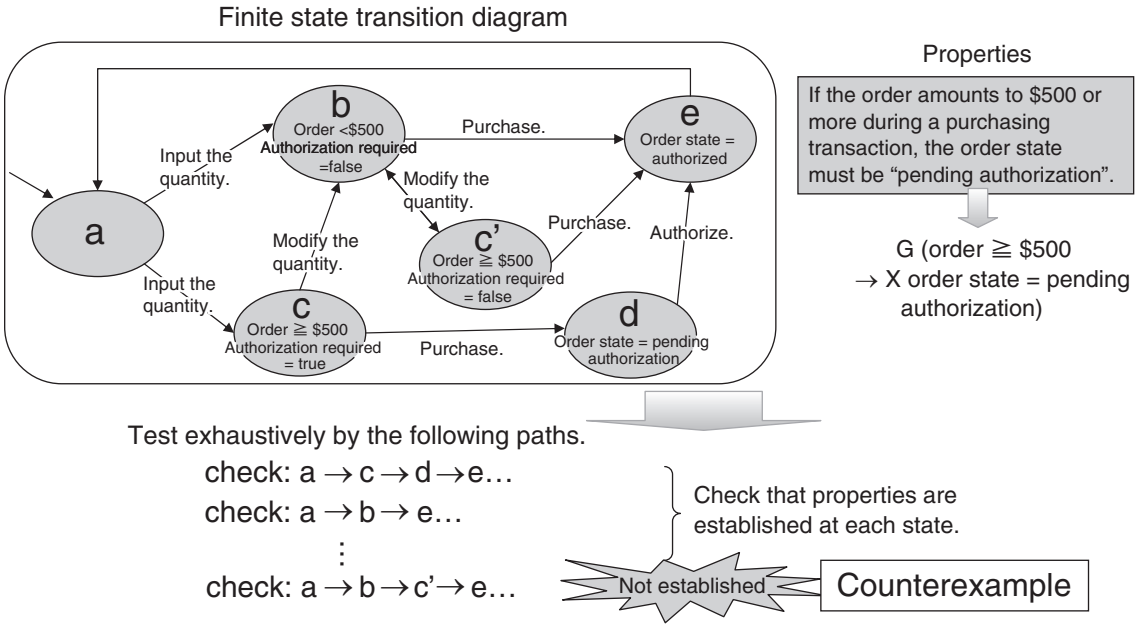

Figure 2
Example of model checking.

developing techniques for automatically verifying whether or not Web applications implemented in Java meet their business specifications.[8),9)]

First, we discuss the issues with conventional test methods. In conventional test methods, the test scenarios and test data are drawn up from external specifications, and tests are run and results are checked almost entirely manually. Consequently, these methods are prone to various sources of human error. For example, some test cases or test data may be overlooked, the number of test cases may be too numerous for complete testing, there may be human errors in the actual tests and results-checking process, and there may be personality-related differences in the way the tests are implemented.

4. New model-checking-based testing technique

4.1 Outline

At Fujitsu Laboratories, we have developed a new technique that addresses the above problems through the use of model checking techniques. An overview of this technique is shown in **Figure 3**. In this technique, the application's external specifications are first expressed in a machine-readable formal representation called a property definition document. Once the external specifications have been defined as properties, the subsequent verification work can be performed automatically. This includes preparing test scenarios and data, performing the actual tests, and checking the results. Rewriting business specifications has a relatively simple one-to-one correspondence, so it is less likely that details will be overlooked, and automating the subsequent tasks eliminates actual tests/checking errors and personality-related differences.

An example of a property definition document for a product sales system is shown in **Figure 4**. Property #1 specifies that the number of stocked units in the stock table must be zero or more, and property #4 specifies that when the Finish input button is clicked, if the customer code on the order registration page is not empty and this customer exists in the customer master table, then the total cost of the order on the next page (order confirmation page) should be the unit

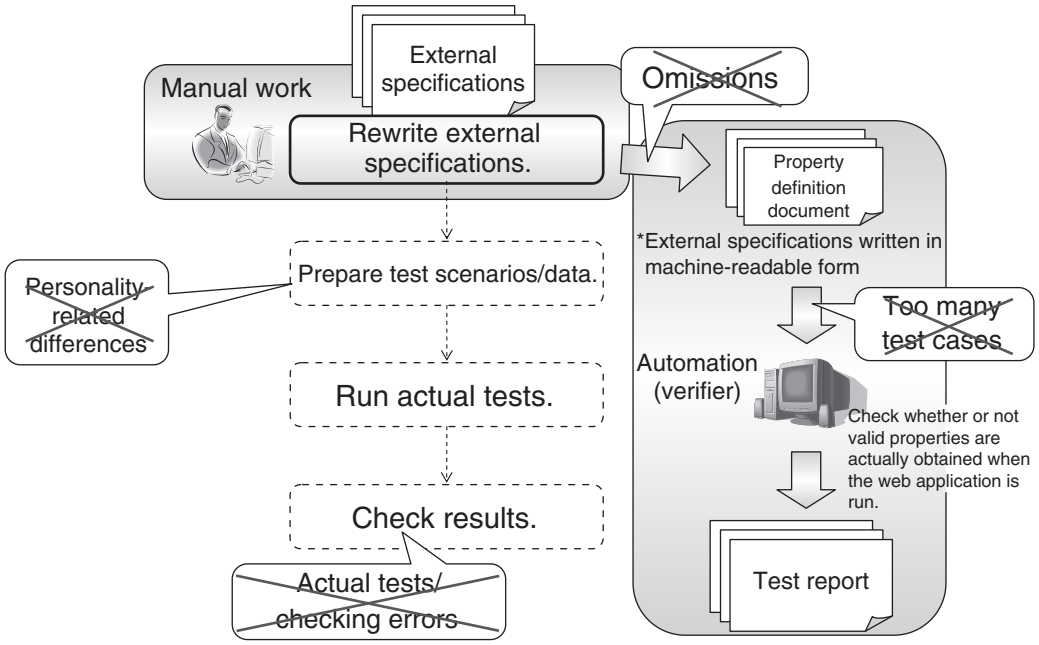

Figure 3
Overview of our technique.

External specifications

Order registration

—	Number of items in stock must always be ≧ 0
Click the “Finish input” button	Display a message in the confirmation message display area and perform the following processing. ・If customer details are missing, display an error message ・If there is no customer code in the customer master table, display an error message ・If there are no errors, calculate [order total] = [unit cost] × [no. of units]

Property definition document

Rewritten by hand

Statements in the form: “If «*precondition*», then «*postcondition*»”

No.	Event	Precondition	Expected result (postcondition)
1	–	–	stock_table.num_items_in stock ≧ 0
2	*Finish* button is clicked.	order_registration_page.customer_code = " "	Error ID =“ZZM9000E”
3	*Finish* button is clicked.	not (order_registration_page.customer_code = " ") not (include(customer mask, customer code from order registration page))	Error ID =“ZZM9001E”
4	*Finish* button is clicked.	not (order_registration_page.customer_code = " ") include(customer mask, customer code from order registration page)	order_confirmation_page.order_total= order_confirmation_page._unit_cost × order_confirmation_page.num_units

Figure 4
Example of property definition document.

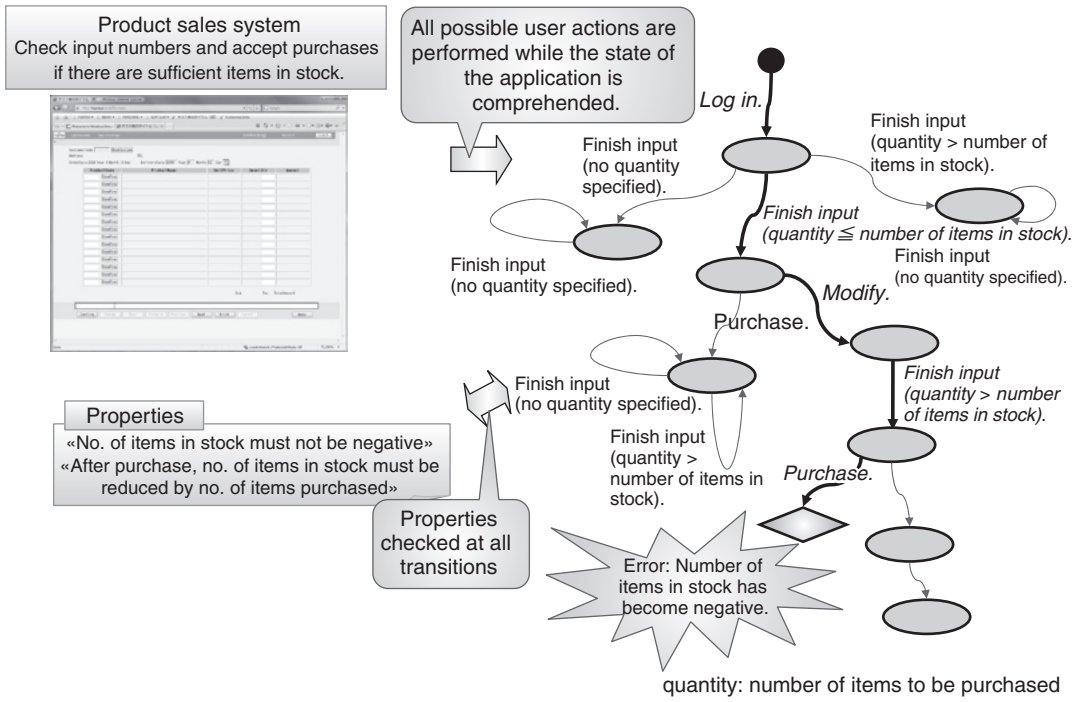

Figure 5
Example of verifying product sales system.

cost multiplied by the number of units.

A verifier is a piece of software that drives a Web application with a variety of different operating procedures and input data on the basis of design information such as a property definition document and screen transition diagram, while checking whether or not its properties (external specifications) are satisfied. If it detects an unmet property, it outputs the sequence of steps leading up to that state as a counterexample.

An example of a verification procedure for a product sales system is shown in **Figure 5**. From the logged-in state, the options for possible user operations and the selection of input data are designated on the basis of the property definition document and design information, and each of these options is run separately. In this case, there are three options: Finish input (no quantity specified), *Finish input* (quantity ≦ number of units in stock), and Finish input (quantity > number of units in stock). Each subsequent state has its own set of executable options that are designated in the same way by repeating the cycle. The software is run exhaustively in this way for all possible combinations of user operations and data. This means that it is verified by being subjected to a thorough set of tests.

While this is going on, continuous checks are made for the occurrence of unsatisfied properties. For example, property #1 in Figure 4 is checked for all actions (state transitions), and #2–4 are checked for all actions where the Finish input button is clicked. In the example in Figure 5, the number of items in stock becomes negative after the *Purchase* button has been clicked in the sequence of clicks indicated by the italicized options. This is a violation of property #1, and the path of options leading to this state is shown by the bold lines.

Two technical aspects of this technique are worth pointing out:

1) Comprehensive searching by Java PathFilter (JPF)[10)]

In this technique, JPF is used as the model checking engine. JPF is an open-source tool developed at the NASA Ames Research Center[11] that has been used as a framework for Java verification tools by a wide range of businesses and research organizations besides Fujitsu Laboratories. It has the following characteristics:

- It incorporates a specially developed Java virtual machine (JVM) that runs the verification byte code. This allows it to access the heap contents, stack, program counter, etc. as state information and select other options by backtracking to any previous state.
- It has an interface for extending the tool's functions. This interface makes it possible to access objects in the heap and modify/extend the search algorithm. For this technique, we have also implemented a function for checking properties on the interface.

2) Reduction in number of items to verify by using application framework profiling

Since JPF uses its own JVM to run the system being verified, the system must be implemented as a single closed Java program. During the verification of a system that accesses a database or a network, these parts must be excluded from the program. We therefore decided to target this technique at business applications developed in Fujitsu's EZDeveloper development framework[12] where the program structure is profiled. In this way, the business logic parts of the program to be verified can be automatically designated, and JPF can be used to check the model by providing stubs to mimic the network or database. This makes it possible to verify full-scale applications by focusing on the parts requiring verification in the above way. In one instance, from a total of 34 316 lines of source code, we were able to restrict the verification to just 9903 lines of business logic.

4.2 Application example

To illustrate this technique, we describe how it was applied to a product sales system developed in EZDeveloper. This example has already been partially introduced in the previous sections, but here we describe the entire application. The main specifications of this system are as follows:

1) On the order page, when a customer selects a product, specifies the number of items required, and clicks the Finish input button, the number of items in stock is checked. If there are sufficient items in stock, the customer is given the option of clicking the Register button.
2) If the customer clicks the Register button on the confirmation page, the order is placed and the customer is returned to the order page.
3) If the customer clicks the Modify button, the customer is returned to the order page without the order being placed.

These screen transitions are shown in **Figure 6**.

When verifying this system with the property definition document shown in Figure 4, we discovered a case that violated the property that the number of stocked units in the stock table must be zero or more (**Figure 7**). This

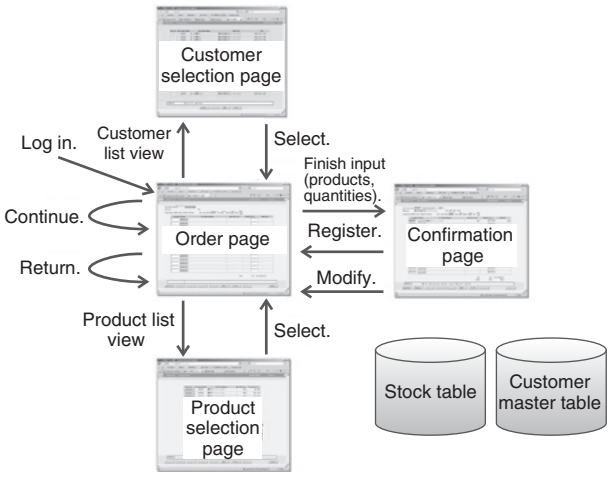

Figure 6
Page transitions in validation example.

fault was detected after the following sequence of operations:

Log in → Finish input (quantity ≦ number of units in stock) → Register → Finish input (quantity ≦ number of units in stock) → Modify → Finish input (quantity > number of units in stock) → Register

In this sequence, a fault unintentionally allowed the Register button to be clicked despite there being insufficient items in stock when the Finish input button was clicked. As a result, the number of items in stock became negative after the Register button was clicked. It was not possible to detect this fault by using typical test cases such as those shown below. The ability of our technique to detect this sort of fault illustrates its strength at exhaustively testing combinations of user operations.

- Log in → Finish input (quantity ≦ number of units in stock) → Register
- Log in → Finish input (quantity ≦ number of units in stock) → Modify
- Log in → Finish input (quantity > number of units in stock) → Modify

... etc.

When this fault was fixed and the verification was repeated, the system passed successfully. An overview of the program states searched by this technique is shown in **Figure 8**. In this example, we not only checked the 30 or so scenarios covered by conventional testing (bold lines in Figure 8), but were also able to automatically verify the equivalent of over 1000 test scenarios while confirming that the specifications in Figure 4 were consistently met.

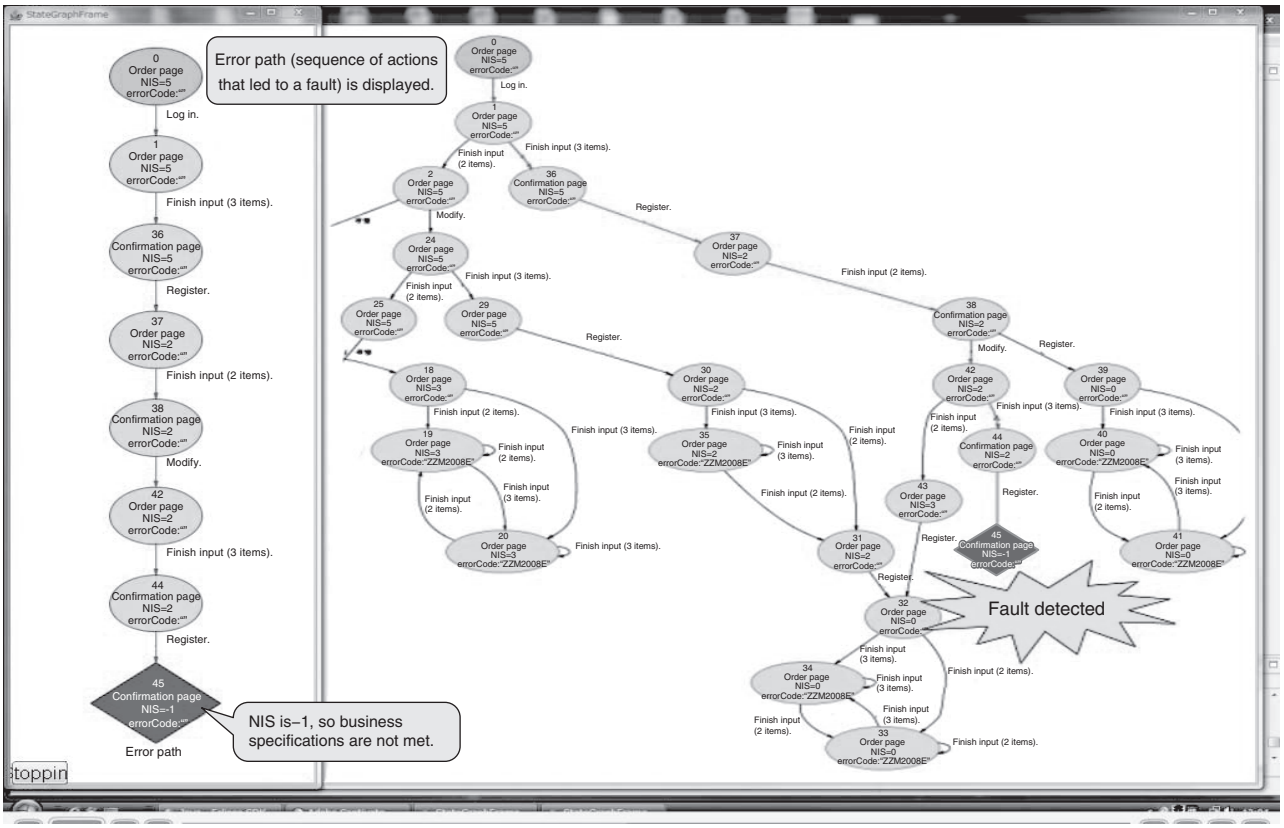

Figure 7
Fault detection (Tool page).

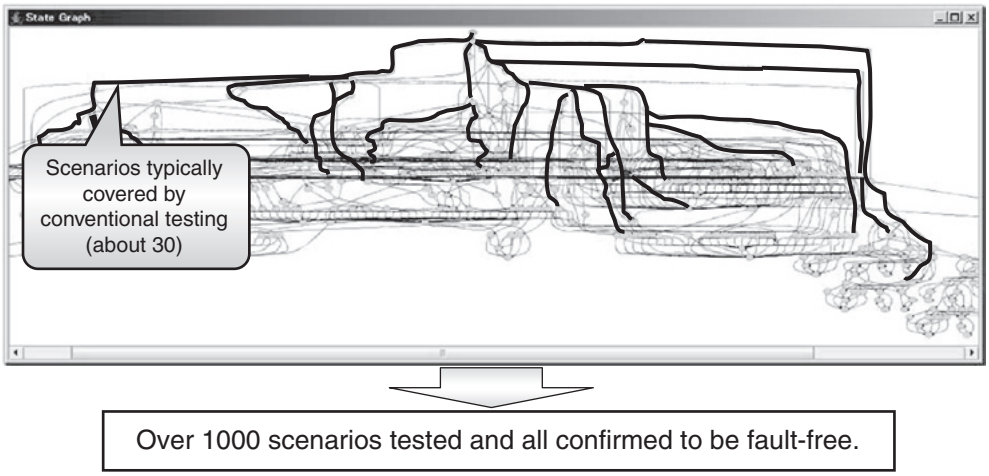

Figure 8
Overview of program states searched by this technique.

4.3 Confirmed benefits and future issues

The technique provides four major benefits over the conventional test techniques. However, there are still a few issues that need to be resolved.

4.3.1 Benefits

In contrast to the issues of conventional test techniques discussed earlier, the benefits of this technique are as follows:

1) Drastically cutting the omission of test cases and data

When verification is performed using this technique, exhaustive tests are performed by varying the combinations of data and user operations on the basis of the property specifications and design values. As a result, the omission of test cases (test scenarios) of the type discussed in section 5 can be almost completely eliminated. On the other hand, the setting of data variations depends on the accuracy with which the property specifications and design information are described, so the omission of a property specification can still lead to the omission of test data.

2) Performing verification with a high test density

Since programs are run automatically, it is possible to verify many more test cases than would be possible by manual testing.

3) Eliminating human error from the running of tests and the checking of results

Since testing and checking are performed automatically, human error can be eliminated from these processes.

4) Excluding personality-related differences

Since the test cases are produced automatically, a uniformly high standard of quality can be achieved by eliminating differences related to the personalities of developers.

4.3.2 Issues

There are three issues that need to be addressed:

1) Describing properties is difficult.

Although external specifications can be converted into a property definition document with a one-to-one correspondence, this is a task that requires specialized formal language skills. To address this issue, techniques and tools should be developed to support the creation of property descriptions (e.g., property editor tools).

2) Test data omissions are still liable to occur.

The designation of data variations as described above is dependent on the accuracy of the statements used to describe properties

and design information. A technique should be developed that incorporates program analysis methods to enable exhaustive verification to be performed without the specifications having to be defined.

3) Only a restricted range of applications can be verified.

This technique can currently verify Web applications developed on a designated framework, but it is necessary to develop a technique for expanding the applicable scope to any ordinary Web application or Java application. Furthermore, in the future we plan to target a wide range of real applications by establishing a development process that combines this technique with conventional test methods by clarifying the separate roles for them.

5. Conclusion

In this paper, we introduced work being done as part of the Production Innovation project at Fujitsu's SE division with the aim of improving software quality based on model checking techniques. We have developed a technique that can be applied to full-scale business software by using JPF as a mechanism for automatically performing exhaustive searches and by combining it with Fujitsu's EZDeveloper development framework. In the future, we intend to make the verification technique even stronger and increase its applicable range, while at the same time pursuing practical advantages by combining it with conventional test methods.

References

1) Ministry of Economy, Trade and Industry: Toward Accelerating IT-Based Productivity Improvement. (in Japanese), IT Frontier Initiative. *http://www.meti.go.jp/press/20070629005/20070629005.html*
2) Construction modes for business systems. (in Japanese), *Nikkei Open Systems*, pp. 118–125 (February 2003).
3) Information Technology Promotion Agency of Japan, Software Engineering Center (IPA/SEC): Software development data white paper 2007. (in Japanese), Nikkei BP, 2007.
4) Special feature: Software is hard. (in Japanese), *Nikkei Electronics*, No. 916, pp. 87–121 (December 19, 2005).
5) Special feature: Formal methods under the spotlight in the drive for bug-free software. (in Japanese), *Nikkei Computer*, No. 657, pp. 60–64 (July 24, 2006).
6) Special feature: Making things more fun. (in Japanese), *Nikkei Computer*, No. 668, pp. 38–53 (December 25, 2006).
7) Ministry of Economy, Trade and Industry: Guidelines for improving the reliability of IT systems. (in Japanese). *http://www.meti.go.jp/press/20060615002/20060615002.html*
8) Fujitsu Laboratories: Fujitsu Develops Software Verification Technology for Practical-use Web Applications. *http://www.fujitsu.com/global/news/pr/archives/month/2008/20080404-02.html*
9) Fujitsu: Basic technique for the automation of testing in Java-based Web application development. *http://jp.fujitsu.com/about/journal/technology/20090401/*
10) Java PathFinder. *http://javapathfinder.sourceforge.net/*
11) NASA Ames Research Center. *http://www.nasa.gov/centers/ames/home/index.html*
12) Fujitsu: Innovative manufacturing and engineering in software development. *http://jp.fujitsu.com/about/journal/publication_number/300/topstory/03.shtml*

Jun Ginbayashi
Fujitsu Ltd.
Dr. Ginbayashi received B.S. and M.S. degrees in Mathematics from the University of Tokyo, Japan in 1981 and 1984 and M.Sc. and D.Phil. degrees in Computing Science from the University of Oxford, U.K. in 1993 and 1996, respectively. He joined Fujitsu Ltd., Tokyo, Japan in 1986, where he has been engaged in software engineering and system development methods and tools. He is a member of IEEE and the Information Processing Society of Japan (IPSJ).

Tadahiro Uehara
Fujitsu Laboratories Ltd.
Mr. Uehara received a B.E. degree in Control Engineering and an 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 in software engineering, especially object oriented technologies and testing technologies for business applications.

Kazuki Munakata
Fujitsu Laboratories Ltd.
Mr. Munakata received an M.E. degree in Information Science from Japan Advanced Institute of Science and Technology, Ishikawa, Japan in 2001. He joined Fujitsu Laboratories Ltd., Kanagawa, Japan in 2005, where he has engaged in research in software engineering, especially software verification and requirements engineering.

Kazuo Yabuta
Fujitsu Ltd.
Mr. Yabuta received an M.S. degree in Electronics from Sophia University, Japan in 1976. He joined Fujitsu Ltd., Tokyo, Japan in 1976, where he has been engaged in software engineering and system development methods and tools. He is a member of Information Processing Society of Japan (IPSJ).