HKU CS Tech Report TR-2005-10

TESTING OBJECT-ORIENTED INDUSTRIAL SOFTWARE WITHOUT PRECISE ORACLES OR RESULTS [*]

by T. H. Tse, Francis C. M. Lau, W. K. Chan,
Peter C. K. Liu, and Colin K. F. Luk

TACCLE tests an automated assembly system in which the expected outcomes cannot be precisely defined and the actual results cannot be directly observed.

About ASM

ASM is the world's largest supplier of assembly and packaging equipment for the semiconductor industry (www.asm.com/about.asp). It manufactures semiconductor assembly equipment and materials (etched and stamped leadframes) used by multinational chip manufacturers, independent IC assembly houses, and consumer electronics manufacturers. The company is also known for its strong commitment to research and product development, supported by more than 550 employees in its Hong Kong and Singapore R&D centers.

From 30% to 50% of the resources in an average software project are typically allotted to testing. Still, inadequate software testing costs industry $22 billion to $60 billion per year in the U.S. alone [8]. We would all spend less if software engineers could use more effective testing methods and automated testing tools. On the other hand, testing is very difficult in real-world projects.

Software testing is commonly accomplished by defining the test objectives, selecting and executing test cases, and checking results [2]. Although many studies have concentrated on the selection of test cases, checking test results is not trivial. Two problems are often encountered:

How to determine success or failure. A test oracle [11] is the mechanism for specifying the expected outcome of software under test, allowing testers to check whether the actual result has succeeded or failed.[1] In theory, test oracles can be determined by the software specification. In practice, however, a specification may provide only high-level descriptions of the system and cannot possibly include all implementation details. Hence, software testers must also rely on domain knowledge and user judgment to evaluate results. Such manual efforts are often error prone [9].

[*] © ACM 2007. This is the authors' version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in *Communications of the ACM 50*, 8 (Aug. 2007), 78–85. http://doi.acm.org/10.1145/1278201.1278210. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works, requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept, ACM Inc., 1515 Broadway, New York, NY 10036, USA, fax +1 (212) 869-0481, or permissions@acm.org.

[1] The word "oracle" is used in the sense of "prophecy".

Hidden results. In engineering projects, even when oracles are present, the results of embedded software may last only a split second or be disturbed by noise or hidden behind a hardware-software architecture, so they are not easily observed or recorded. Moreover, the observation or recording process may introduce further errors or uncertainties into the execution results.

It is therefore impractical for testers of engineering projects to expect to have predetermined, precise test oracles in every real-world application. How to capture and evaluate test results poses another problem. Automating the testing process amid these uncertainties is especially difficult.

Here, we share our experience addressing these issues in a technology-transfer project funded in 2002–2004 by ASM Assembly Automation Ltd. and the Innovation and Technology Commission in Hong Kong, examining the application of advanced testing methodologies to alleviate these problems (see the sidebar "About ASM").

Methodology and Tool

ASM's assembly equipment is supported by embedded software developed in C11 and built with extensive error-avoidance and recovery features. The software specifications are extracted from technical drawings of mechanical and electronic hardware designed by process engineers using an in-house technique.

We describe our testing methodology and tool for the technology-transfer project. The system consisted of three components:

- A specification editor to capture the requirements defined by process engineers;
- Automated black-box testing that bypasses the need for precise oracles; and
- Semiautomated integration of black- and white-box testing to check the consistency of machine and human evaluations of the test results.

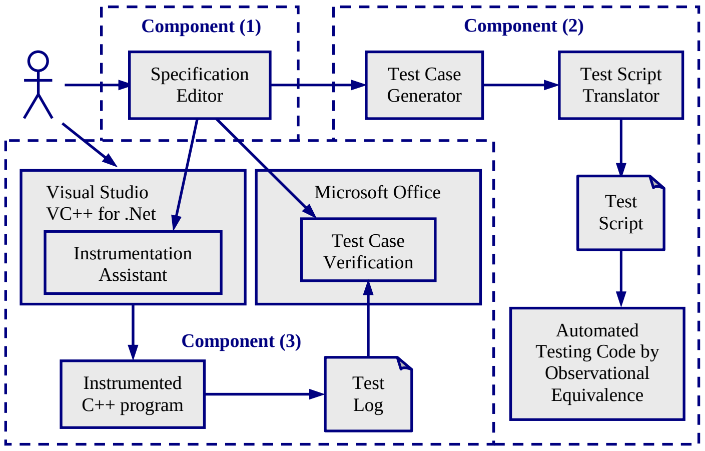


Figure 1. Workflow of the testing tool.

These components are outlined in Figure 1 and described in the following sections:

Component (1). Specification editor. The front end of the testing tool is an XML-based editor designed to capture the stations and actions specified in the timing diagrams, as well as their relationships with the classes and methods in the implemented programs. It captures the following:

- Station types;
- Realization and organization of different types of equipment component;
- Station interactions;
- Object classes implemented according to the specified stations;
- Software messages invoking services that implement the actions and triggers in the timing diagrams; and
- State enquiry functions of the implemented classes.

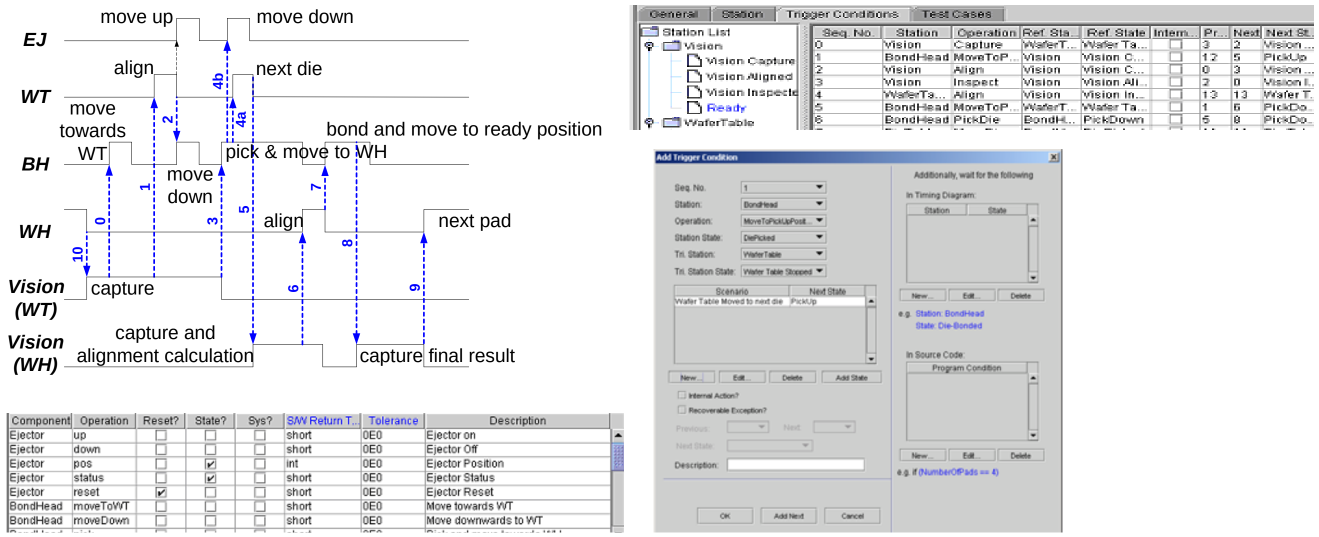
Our specification editor keeps the algebraic specifications internal to the testing tool and transparent to the user.

Timing Diagrams	Algebraic Specifications
Stations	Classes
Interactions and cycles	Equations
Path selection conditions	Conditions of equations
Actions	Methods
State enquiry functions	Observers
Variants of operations	Fully expanded into classes, methods, and equations.

Table 1. Mapping of entities between timing diagrams and algebraic specifications.

The editor also captures other test parameters (such as the maximum number of test cases, the classes to be tested, and the classes accessible by testers).

The internal model used by the specification editor is a communicating finite-state machine



Seq. No.	Station	Operation	Ref. Sta...	Ref. State	Intern...	Pr...	Next	Next St...
0	Vision	Capture	WaferT...	Wafer Ta...	☐	3	2	Vision ...
1	BondHead	MoveToP...	Vision	Vision C...	☐	12	5	PickUp
2	Vision	Align	Vision	Vision C...	☐	0	3	Vision ...
3	Vision	Inspect	Vision	Vision Ali...	☐	2	0	Vision I...
4	WaferTa...	Align	Vision	Vision In...	☐	13	13	Wafer T...
5	BondHead	MoveToP...	WaferT...	Wafer Ta...	☐	1	6	PickDo...
6	BondHead	PickDie	BondH...	PickDown	☐	5	8	PickDo...

Add Trigger Condition

Seq. No.: 1
Station: BondHead
Operation: MoveToPickUpPosit...
Station State: DiePicked
Tri. Station: WaferTable
Tri. Station State: Wafer Table Stopped

Scenario	Next State
Wafer Table Moved to next die	PickUp

☐ Internal Action?
☐ Recoverable Exception?
Previous:
Next:
Next State:
Description:

Additionally, wait for the following

In Timing Diagram:

Station	State

e.g. Station: BondHead
State: Die-Bonded

In Source Code:

Program Condition

e.g. if (NumberOfPads == 4)

Component	Operation	Reset?	State?	Sys?	S/W Return T...	Tolerance	Description
Ejector	up	☐	☐	☐	short	0E0	Ejector on
Ejector	down	☐	☐	☐	short	0E0	Ejector Off
Ejector	pos	☐	✓	☐	int	0E0	Ejector Position
Ejector	status	☐	✓	☐	short	0E0	Ejector Status
Ejector	reset	✓	☐	☐	short	0E0	Ejector Reset
BondHead	moveToWT	☐	☐	☐	short	0E0	Move towards WT
BondHead	moveDown	☐	☐	☐	short	0E0	Move downwards to WT

Figure 2. Representations of timing diagram (counterclockwise from top-left). Sample timing diagram provided by engineers, definition of operations for stations, details of a transition, and state transition summary.

(CFSM) model [3] that represents timing diagrams (see Figure 2 top left). Since our testing methodology is built on algebraic specifications [6], the editor further translates the CFSM model into an algebraic representation. The table here maps the entities between timing diagrams and algebraic specifications; for example, stations and actions in timing diagrams bear the same meaning as classes and methods in algebraic specifications. Figure 2 includes two tabular views of the testing tool, capturing the operational definitions of the classes in the software and the temporal properties of the timing diagrams; the bottom-right screenshot shows how the tool captures the transition condition between two states. The purpose is to bridge any gap between a specification and its implementation. We appreciate that most software developers are not completely comfortable with formal algebraic specifications because they involve unfamiliar mathematical concepts and manipulations. Our specification editor keeps the algebraic specifications internal to the testing tool and transparent to the user.

Component (2). Black-box testing at the class and cluster levels. Timing diagrams are high-level specifications that define the interactions among stations. Although the diagrams show the operations that update the attributes of a station and trigger further actions in other stations, operational details (such as how they achieve the desired results) are not defined in full. Furthermore, lower-level operations (such as exception handling and transitions among internal states of a station) may not be described. Thus, timing diagrams cannot serve as a precise oracle for testers.

On the other hand, objects in the implemented system must preserve the high-level behaviors specified by the timing diagrams irrespective of exceptional conditions and other implementation decisions. For example, a pick arm in an automated assembly system may encounter a problem when attempting to put a die in place. As a result, it should place the die in the discard bin and then move itself back to the home position. Another pick arm may not encounter a problem and should therefore place the die in the standard collection bin and then move itself back to the home position. Once the two pick arms return to their home positions, they should forget about any abnormal incidents and behave exactly the same way from now on. We say these pick arms are now "observationally equivalent."

To test whether two objects are observationally equivalent, we apply an advanced testing methodology known as object-oriented software Testing At the Class and Cluster LEvels, or TACCLE [4]. As the name suggests, TACCLE enables software engineers to test each individual class independently, then test the interactions among classes, as described in the following sections:

Class-level testing. A station specified in the timing diagram is implemented as a class in the embedded system. We pick up two instances of a station specified in the timing diagram to be observationally equivalent despite variations in operational detail (such as exceptional conditions). Based on intuition, we must execute the two objects in the implemented system that correspond to these two specified instances and test whether they are indeed observationally equivalent. Based on similar intuition, given two instances of a station specified as observationally nonequivalent, we must also test whether the corresponding implemented objects are observationally nonequivalent. However, we note from technical fundamentals that two objects are observationally equivalent if and only if they can be subjected to any sequence of operations and give identical results.[2] [1] In practice, the number of possible operation sequences may be infinite, and hence testers may have to devote an impossible amount of time checking the observational equivalence of even a single pair of objects.

We used observational equivalence to substitute for the need for precise oracles and results, then used two simple criteria to substitute observational equivalence.

Fortunately, we have mathematically proved [4] that the following criteria can reveal exactly the same set of failures as the testing of observational equivalence but are much simpler to apply in practice:

Criterion A. If two instances of a station in a timing diagram are specified to have equivalent histories of operations, then the final observable attributes of their implemented objects should have the same values; and

Criterion B. If the final observable attributes of two instances of a station in a timing diagram are specified to have different values, then the final observable attributes of their implemented objects should also have different values.

Consider again the two pick arms that place the dies in appropriate bins. To test whether they are observationally equivalent, we need to subject each one to every possible sequence of operations and check their results. We resolve to use Criteria A and B instead. To test according to Criterion A, for instance, we execute two operation sequences to generate two instances of the pick arm, one with a *problem, the other without a problem. The state enquiry function will check the final observable* attributes of the two instances (such as the locations of the pick arms and their reset values). Whenever the respective values are different, the tool reports an error in the implementation.

Cluster-level testing. At the cluster level, we test the interactions among a group of stations rather than within only one station. We apply a sequence of operations to generate a cluster of stations with a specific state. We also find an alternate operation sequence that generates another cluster with the same state. We then check the observable attributes of the implemented clusters to verify whether they have indeed reached the same state.

Consider a simple industrial example involving the following three stations: a wafer table, a bonding unit, and a vision system. The wafer table triggers the vision system to take a picture of the next die in the queue. When the picture is ready, the vision system notifies the wafer table to move the die to the bonding position. To minimize latency, the bonding unit also moves to this position so it can pick up the die when the die reaches the appropriate place.

There may be a small chance, however, that the die in the queue is missing, the movement may differ from the operational standard, or the bonding unit may accidentally block the photography process. In such circumstances, the wafer table will advance the next die in the queue, the bonding unit will move back to a safe position, and the vision system will take another picture. In short, the system should completely ignore a defective cycle and continue to process the next die as if the problematic cycle did not exist.

To test whether the system really implements this behavior, we generate two groups of objects for these stations, one with and one without the problematic cycle. We compare their respective results (such as arm positions) through state enquiry functions. Since the two clusters are supposed to have equivalent histories of operation, the difference in values observed at their final states should be within acceptable limits of natural variation. The upper part of Figure 3 outlines sample test code generated by the tool.

Test case generation and test script translation. Component (2) of the test tool consists of a test-case generator and a test-script translator. The test-case generator produces equivalent pairs of stations or clusters by following the TACCLE methodology described earlier. These test suites are represented in XML format. The test-script translator accepts the XML file and, after packaging it with initialization preambles and error-reporting episodes [5], translates the conceptual test cases into test scripts for various standards (such as Microsoft C11 .NET, Java, and company-specific XML). The test scripts are also designed to meet industrial and company-specific programming standards.

[2] "Observational equivalence" is a standard term used in finite state machines and related areas [8, 10]. Two states are observationally equivalent if and only if they produce the same results when subjected to the same sequence of transitions.

The C11 or Java test scripts can then be compiled with the original source code. Following a reboot of the equipment, the CFSM-related behaviors of the object-oriented classes can be tested automatically.

Any compromise in quality. When designing Component (2), we used observational equivalence to substitute for the need for precise oracles and results, then used Criteria A and B to substitute observational equivalence. Process engineers might want to know whether there is any compromise in quality. We provide the following two observations:

- We have mathematically proved that the use of Criteria A and B reveals exactly the same set of failures as the testing of observational equivalence, even though this property may appear counterintuitive to some engineers. Hence, there is no concession as far as effectiveness is concerned; and
- On the other hand, there is a limitation in the testing of observational equivalence because it does not check whether a specific attribute of the system has a certain value at a certain moment.

With respect to the second observation, we note that the need to test observational equivalence arises from practical constraints. In object-oriented systems, not all attributes are visible. The timing diagrams in the ASM project do not, for instance, provide sufficient information for testers to define a precise oracle. Furthermore, the software is embedded, and the outputs from a station are hidden, only to be consumed by other stations with the same embedded software. Hence, the testing of observational equivalence provides a useful (but incomplete) technique in the absence of precise oracles or results.

Fortunately, real-world systems must communicate with the external

Comparing Related Testing Methodologies

State values of objects were used in [12] as test oracles in their experiments on class testing. However, [15] pointed out that "oracles built by violating object encapsulation [so as to access hidden states] may result in differences in behavior between what to test and what to use," and hence "an effective alternative to violating encapsulation is to provide a way of determining whether two objects are equivalent." The ASTOOT approach, or A Set of Tools for Object-Oriented Testing, introduced a technique for testing pairs of objects that are expected to be equivalent or nonequivalent in their behavior. Unfortunately, the theory discussed in [14] is not without flaws. The "black and white" approach [13] improved on the idea of ASTOOT to reliably generate object pairs that are equivalent. It was further enhanced into the TACCLE approach [4], proving mathematically that the difficult tasks of testing observational equivalence and nonequivalence can be simplified into easier and more viable tasks. We made full use of the TACCLE approach in the ASM project.

The authors of [1] checked the consistency of test logbooks against test specifications. Nevertheless, for embedded software that controls the movement of delicate hardware, a logbook entry recorded by a program may be misaligned with hardware behavior. We enhanced the checking of test logs by also taking user observations into account.

User Experience

The training of ASM software engineers to operate the testing tool involved two iterations. It took two days for the average engineer to be trained on the working principles of the testing tool and another two days to define the timing diagrams and observable attributes of a typical machine. After reviewing the input models thus produced, we retrained the engineers in areas in which their concepts did not align properly with those of the testing tool. It took half a day to complete the retraining and another half day to rectify the input models.

Based on the input models, it took less than 30 seconds to generate hundreds of test cases and overnight to generate a million test cases on a standalone Intel Celeron machine with 400Mz CPU running Windows 2000. On identifying any failure, the system generates a test report in Microsoft Excel within 10 seconds.

Users reported being "satisfied" with the tool and find the "knowledge and experience gained ... helpful." On the other hand, at the beginning of the technology-transfer project, there was resistance from several users regarding the introduction of new testing concepts, deployment of a new testing tool, and appointment of an independent consultancy team to "help" verify the correctness of their software. Although psychological resistance occurs in many work-related contexts, this point should nevertheless be noted.

12. Briand, L., di Penta, M., and Labiche, Y. Assessing and improving state-based class testing: A series of experiments. *IEEE Transactions on Software Engineering 30,* 11 (Nov. 2004), 770–793.
13. Chen, H.Y., Tse, T.H., Chan, F.T., and Chen, T. Y. In black and white: An integrated approach to class-level testing of object-oriented programs. *ACM Transactions on Software Engineering and Methodology 7*, 3 (July 1998), 250–295.
14. Doong, R.-K. and Frankl, P.G. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology 3,* 2 (Apr. 1994), 101–130.
15. Pezze, M. and Young, M. Testing object-oriented software. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)* (Edinburgh, U.K.). IEEE Computer Society Press, Los Alamitos, CA, 2004, 739–740.

world, so oracles and results are not totally absent. We may integrate Component (2) with conventional testing or other techniques to achieve a total test plan. In Component (3), we illustrate how we can take user observations and evaluations into account to make the testing more comprehensive.

Component (3). Program instrumentation and consistency checks. We cannot rely fully on observational equivalence for software testing. Suppose, for instance, the software of a controller has a fault such that it sends out a "danger" signal when the situation is safe and a "safe" signal when the situation is dangerous. Suppose the software of the bonding machine in the assembly system has a similar fault when interpreting the "danger" and "safe" signals it receives. In this case, Component (2) of our testing tool cannot identify behavioral inconsistencies. Although human users can observe and identify the error, they may still make mistakes, especially when testing under tight time and budgetary constraints. It would be useful to have an additional mechanism to check the consistency between machine and human interpretations. Component (3) supports this by means of program instrumentation and consistency checks among timing diagrams, test results, and human observations.

Real-world situations in the assembly system are much more complex than this. Consider, for instance, a scenario involving the production of a high-quality die. Process engineers perceive that the bonding unit should pick up the current die at the central spot, place it on a bin, and then bond it. The actual implementation involves elaborate exception-handling routines. The bonding unit can pick up the die only after the wafer table has made minor adjustments that match the die but before the wafer table starts repositioning itself to fit the next die. The bonding unit is also required to place the die at an appropriate location while not blocking the vision system from capturing an image for analysis. To ensure a comprehensive consistency check, Component (3) conducts test log analysis (similar to [1]) to verify whether:

- The actual behavior detected by the instrumented program is different from the specified behavior;
- The actual behavior observed and recorded by the user is different from the specified behavior; and
- The user evaluation of consistency is different from the evaluation by the consistency-checking mechanism.

To select scenarios for testing, process engineers can use the specification editor to highlight appropriate clusters of stations in timing diagrams. Relevant test code for recording behavior is generated automatically and added to the original source code of the application. After simple manual operations to define appropriate locations for state enquiries, the resulting program probes the observable attributes of the objects via state enquiry functions. Engineers then observe the running of the implemented objects and record the actual operation sequences into an electronic logbook built into the system. They also add their own evaluation of whether the resulting execution is consistent with the specified scenario.

Apart from test-log analysis, Component (3) has been integrated with Microsoft products, one of the software development platforms used by ASM. It extracts from the specification editor an annotated table that maps the stations of the timing diagrams to actual classes in the source code. The information is incorporated into a .NET add-in, so software engineers can drag and drop instrumentation code to the source code of the project (the lower part of Figure 3). The instrumented code consists of C11 macros. Simple configuration directives are built into Component (3) to turn the instrumentation on or off, so production software is delivered without additional effort to remove the instrumented statements. The inputs of user observations, test analysis, consistency checks, and error reports are implemented in Microsoft Office. Other nontesting features (such as printing, version control, and collaborations) are also handled by various Microsoft products.

Conclusion

We have highlighted the problems software testers can face in industrial projects where precise test oracles or test results may not be available. We have applied advanced testing techniques in our TACCLE methodology to a real-world engineering project for ASM. The results are encouraging. Despite the oracle problems, all the stations specified by timing diagrams can be tested via observational equivalence and the results verified automatically or semiautomatically. At the same time, the testing tool renders the abstract mathematical concepts and formal algebraic specifications behind observational equivalence transparent to software developers and testers.

The study produced two key insights: First, the notion of testing observational equivalence and bypassing the need for oracles is important in test automation for industrial projects where it is impractical to define a precise relationship between the specification and the software under test. And, second, even imprecise additional information from the application domain may be beneficial for the purpose of enhancing software quality through instrumentation and consistency checks.

References

1. Andrews, J.H. and Zhang, Y. General test result checking with log file analysis. *IEEE Transactions on Software Engineering 29,* 7 (July 2003), 634–648.
2. Beizer, B. *Software Testing Techniques.* Van Nostrand Reinhold, New York, 1990.
3. Brand, D. and Zafiropulo, P. On communicating finite-state machines. *Journal of the ACM 30,* 2 (Apr. 1983), 323–342.
4. Chen, H.Y., Tse, T.H., and Chen, T.Y. TACCLE: A methodology for object-oriented software testing at the class and cluster levels. *ACM Transactions on Software Engineering and Methodology 10,* 1 (Jan. 2001), 56–109.
5. Fetzer, C., Felber, P., and Hogstedt, K. Automatic detection and masking of nonatomic exception handling. *IEEE Transactions on Software Engineering 30,* 8 (Aug. 2004), 547–560.
6. Goguen, J.A. and Malcolm, G., Eds. *Software Engineering with OBJ: Algebraic Specification in Action.* Kluwer Academic Publishers, Boston, 2000.
7. Milner, R. *Communication and Concurrency. Prentice Hall International Series in Computer Science.* Prentice Hall, Hemel Hempstead, Hertfordshire, U.K., 1989.
8. National Institute of Standards and Technology. *The Economic Impacts of Inadequate Infrastructure for Software Testing. Final Report.* Gaithersburg, MD, 2002; www.nist.gov/director/progofc/report02-3.pdf.
9. Peters, D.K. and Parnas, D.L. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering 24,* 3 (Mar. 1998), 161–173.
10. van Gabbeek, R.J. and Weijland, W.P. Branching time and abstraction in bisimulation semantics. *Journal of the ACM 43,* 3 (May 1996), 555–600.
11. Weyuker, E.J. On testing non-testable programs. *The Computer Journal 25,* 4 (Nov. 1982), 465–470.

T. H. TSE (thtse@cs.hku.hk) is a professor in the Department of Computer Science of The University of Hong Kong.

FRANCIS C. M. LAU (fcmlau@cs.hku.hk) is a professor in the Department of Computer Science of The University of Hong Kong.

W. K. CHAN (wkchan@cs.cityu.edu.hk) is a lecturer in the Department of Computer Science of City University of Hong Kong. Part of the project was conducted when he was with The University of Hong Kong.

PETER C. K. LIU (peter.liu@asmpt.com) is the Chief Technology Officer of ASM Assembly Automation Ltd., Hong Kong.

COLIN K. F. LUK (colin.luk@asmpt.com) is the Director of Technology (Software) of ASM Technology Singapore (Pte) Ltd., Singapore.

This project was supported in part by a matching grant from ASM Assembly Automation Ltd. and the Innovation and Technology Commission in Hong Kong (project no. UIM/77), a grant from the Research Grants Council of Hong Kong (project no. 714504), and a grant from City University of Hong Kong (project no. 7200079).