To appear in *Proceedings of the 2nd ACIS Annual International Conference on Computer and Information Science* (*ICIS 2002*), International Association for Computer and Information Science, Mt. Pleasant, Michigan (2002)

An Overview of Integration Testing Techniques for Object-Oriented Programs *

W. K. Chan
Department of Computer Science and Information Systems
The University of Hong Kong
Pokfulam Road, Hong Kong
wkchan@csis.hku.hk

T. Y. Chen
School of Information Technology
Swinburne University of Technology
Hawthorn 3122, Australia
tychen@it.swin.edu.au

T. H. Tse †
Department of Computer Science and Information Systems
The University of Hong Kong
Pokfulam Road, Hong Kong
tse@csis.hku.hk

Abstract

Object-oriented programs involve many unique features that are not present in their conventional counterparts. Examples are message passing, synchronization, dynamic binding, object instantiation, persistence, encapsulation, inheritance, and polymorphism. Integration testing for such programs is, therefore, more difficult than that for conventional programs. In this paper, we present an overview of current work on integration testing for object-oriented and/or concurrent programs, with a view to identifying areas for future research. We cover state-based testing, event-based testing, fault-based testing, deterministic and reachability techniques, and formal and semi-formal techniques.

Keywords: Integration testing, object-oriented programs, concurrent programs.

1. Introduction

Object-oriented programming is considered as the paradigm of choice today. However, most testing techniques were originally developed for the imperative programming paradigm, with relative less considerations to object-oriented features such as message passing, synchronization, dynamic binding, object instantiation, persistence, encapsulation, inheritance, and polymorphism. Objects may interact with one another with unforeseen combinations and invocations. The testing of concurrent object-oriented systems has become a most challenging task.

Object-oriented programs can be tested at four levels, namely the algorithmic level, class level, cluster level, and system level [1, 2]. At the algorithmic level, individual methods are tested separately, so that conventional testing techniques can be applied without much problem. At the class level, the objective is to verify the integrity of a class by testing it as an individual entity. These two levels have been widely studied [1, 3, 4]. The cluster level is concerned about the integration of classes. As the functionality of individual classes has already been verified, the focal points are usually placed on the synchronization of different concurrent components as well as inter-class method invocations. At the system level, interactions among clusters are tested.

This paper reviews common techniques for program testing at the cluster level. They include state-based testing, event-based testing, fault-based testing, deterministic and reachability techniques, and formal and semi-formal techniques.

*This work is supported in part by grants of the Research Grants Council of Hong Kong (Project Nos. HKU 7033/99E and 7029/01E), a grant of the Innovation and Technology Fund (Project No. UIM/77), and a research and conference grant of The University of Hong Kong.

†**Contact author.**

2. Common Integration Testing Techniques

In this section, we review some of the common approaches in the integration testing of object-oriented programs.

2.1. State-Based Testing

State-based testing techniques rely on the construction of a finite-state machine (FSM) or state-transition diagram to represent the change of states of the program under test. For integration testing, the construction of a global FSM may become unmanageable and subject to the state-explosion problem. Conventional techniques for concurrent programs treat a program as a static set of communicating components and model it as deterministic or non-deterministic FSMs communicating with one another. They systematically arrange the components into an FSM hierarchy by reducing composite FSMs at each level by means of abstraction (thus hiding unimportant or internal events), and classifying interaction statements as local and global points [5]. Alternatively, they may prune out unnecessary or infeasible state transitions, such as by employing interface processes [6]. Some methods even store the removed details in the edges [7]. The resultant graphs often consist of much fewer nodes and edges than the flattened composition of all FSMs, so that it will be feasible to traverse all nodes and edges. In this way, the state-explosion problem can be alleviated. The models may then be verified for selected properties such as deadlock free or livelock free. From the testing perspective, test cases can also be selected based on graph traversal. Strategies such as all-synchronization-pair coverage have been proposed. A potential research direction is the study of more advanced techniques in graph minimization.

Object-oriented programs are often treated as a special kind of concurrent program. However, object-oriented features such as object instantiation, persistence, and inheritance may impose additional constraints to the applicability of this concept. Classes are instantiated dynamically as objects. The states of the objects may evolve beyond the life of a program. Objects may interact with one another in unforeseen combinations and invocations. The behavioral properties of certain objects in a class may differ from those of other objects in the same class, depending on the time and type of instantiation. New testing techniques will need to be developed to address these features.

2.2. Event-Based Techniques

Instead of using the state-based approach, the synchronization sequence for a concurrent program can also be viewed as relationships between pairs of synchronization events. Techniques based on temporal relationships among synchronization events, such as message sequence constraints or temporal logic, have been studied. A merit of these approaches is that they support the analysis of event sequencing requirements without having to cater for the states of the program or components.

In CSPE [8], for instance, relationships between pairs of events can be classified into three types, namely "always valid", "possibly valid", and "never valid". They represent, respectively, situations where the first event should be, may be, and should not be followed by the second event. The "possibly valid" relationship is further classified into "possibly true" and "possibly false". Further constraints can be constructed using the transitivity of the "imply" operator. Suppose, for instance, it is "always valid" that an event P is followed by an event Q, and "never valid" that event Q is followed by an event R. We can conclude that event P followed by event R is "never valid". This negative relationship can be used to check the output. A coverage criterion can be specified to cover the "always valid" and "possibly valid" situations for all event pairs identified directly or indirectly, and test output can also be checked for non-violation of the "never valid" situation. Common methods such as random testing and partition testing can be used to generate test cases. Further research will be required to study whether event-based techniques can equally be applied to integration testing for object-oriented programs.

Every object instantiated by an object-oriented program can potentially be a concurrent component. This poses a new challenge for software testers. For example, when we analyze a static unit, namely a class, we may conclude that any pair of events can be "possibly valid". When we analyze a dynamic unit, namely an individual object of that class, however, we may find that every pair of events can only be "always valid" or "never valid". A faulty implementation that permits "possibly valid" situations in individual objects may not be identified easily. The situation will be worsened if overriding is allowed. For instance, a constraint may be "possibly valid" for objects in the superclass but "always valid" for objects in a particular subclass.

2.3. Testing against Formal Specifications

A lot of research has been done using formal specifications for the testing of object-oriented programs at the class level. For example, Doong and Frankl [2] and Chen et al. [1, 4] have proposed to test behavioral equivalence of two objects in a class by applying algebraic specification techniques [9]. However, relative little work has been conducted for integration testing.

Contract [10] is a formal language for specifying the behavioral dependencies and interactions among objects of different classes. Such behavioral properties are defined using "message-passing rules" (mp-rules). Chen et al. [1] have proposed to apply Contract specifications to integration testing for object-oriented programs. Testing procedures have been defined for individual mp-rules as well as composite mp-rules. The authors have also made use of the similarities between the control mechanism of these mp-rules and the inference engine of Prolog. Hence, they have implemented two automatic testing tools for integration testing using Arity/Prolog.

Different formal paradigms have different advantages and disadvantages in supporting the testing of object-oriented programs. A useful research direction is to explore whether integrated formal paradigms is a viable option. Examples are to combine object-orientation with net-based specifications such as finite-state machines and Petri nets, model-based specifications such as Object-Z [11], and process algebra such as CSP [12]. For instance, Chang et al. [13] generate test scenarios based on FSM specifications and Object-Z; Hong and Bae [14] defined a hierarchical object-oriented Petri net; and Smith and Derrick [15] combined data structure modeling in Object-Z and communication behavioral descriptions in CSP. In theory, such integrated frameworks should be very useful for a comprehensive testing of all the features of object-oriented programs. In practice, however, the complexity of the combined models may be a serious concern for software testers.

2.4. Deterministic and Reachability Testing Techniques

Arranging synchronization orders is a classical deterministic testing approach that usually involves the selection of synchronization sequences and forced execution of a program according to the desired path.

Eraser [16, 17] is a technique based on the concept of locksets to detect a datarace, that is, a violation of the mutual exclusion principle for inputs to a multi-threaded program. A shared variable is protected by a non-empty set of locks. If the lockset can be reduced to an empty set, it will be a sign of violation. The reduction process can be achieved by comparing the current lockset of the variable against the latest lock protection applied to that variable. Any lock in the current lockset that is not protected by the latest access to the variable will be eliminated. Eraser enumerates all the possible synchronization orders for a given input to check whether the lockset is empty. By checking all the synchronization orders, it can also verify whether a deadlock will occur for that input. Brening [16] has implemented the approach at the Java bytecode level. Another dynamic datarace detection mechanism has also been reported [18]. A potential area for further research is a more general lock-protection scheme.

Similarly, Seo et al. [19] have used the synchronization sequence encoded as a dependency table to generate a precedence graph, translated it into an automaton class, and then geared it to a Java program. The equivalent automaton generator produces equivalent synchronization sequences, based on which the system will produce equivalent objects. Unlike most of the forced deterministic testing techniques, the execution order of Java threads is controlled via an instrumented class known as a Controller in the program. The Controller allows a thread to execute if the event requested by the thread is acceptable by the automaton. Otherwise it blocks the thread using the Java method *wait*(). By permitting or blocking the execution of threads, the Controller guides the program to desirable paths, which are generated to be equivalent by the automaton generator in the previous step. The result of equivalent sequences will be checked against the corresponding permitted and forbidden events. Put in the process algebra terminology, the checking attempts to confirm whether a trace of a process, and the refusal set of that trace between the equivalent sequences, are not the same. Unlike Eraser, it requires the program specification to generate a dependency graph. Moreover, this approach generates a global automaton. The complexity of the FSM thus generated has not been discussed in the literature.

Similar techniques to re-arrange the execution order of synchronization, by means of a process priority scheduler not within the control of the program, has also been widely proposed and used in deterministic and reachability testing of concurrent systems [20]. The latter uses forced deterministic testing primarily for efficiency, which is always an important factor for consideration in software

testing.

2.5. Fault-Based Techniques

In mutation testing, a set of mutation rules is used to change a program, thus creating alternate programs known as mutants. The mutants are tested to verify that different results are obtained from the same test cases. The technique has been applied successfully to imperative programs such as Fortran and C. Tools such as Mothra [21] have been developed. There has also been research [22] on interface mutants for integration testing of conventional programs. There are also proposals, such as [23], which combine state-based techniques with mutation testing.

Kim et al. [24] suggest, however, that conventional mutation systems may not be effective for identifying faults specific to object-oriented programs. They propose new rules for object-oriented features such as message passing, encapsulation, inheritance, and polymorphism, and measure their effectiveness through empirical studies. Whether the approach can be applied to integration testing for object-oriented programs require further studies.

2.6. UML-Based Techniques

UML [25] is a semi-formal modeling language that is popular in object-oriented software development. It includes class diagrams, activity diagrams, sequence diagrams, collaboration diagrams, state diagrams, and others. Class diagrams describe general relationships amongst classes. Activity, sequence, collaboration, and state diagrams describe the interaction of objects at different levels of abstraction and from different points of view. For instance, activity diagrams may illustrate a use case from the users' point of view while sequence diagrams express interactions among objects in greater detail. For integration testing, these diagrams help express relationships among method invocations and will, therefore, be useful sources of information.

TOTEM [26] has been proposed to test object-oriented systems based on UML specifications. The main test plan consists of use case sequences. Each use case is associated with a sequence diagram. The diagram is translated formally into a regular expression. Each term in the regular expression represents either a use case scenario or, in the presence of iteration symbols, a set of scenarios. Associated with each path in the sequence diagram is a guard condition expressed in an object constraint language. The exact operation sequences to be executed for each term, including inter-dependencies, are also identified. For example, operation sequences for iterations include a maximum number of executions, an intermediate number of executions, once only, and none at all. The testing oracles are specified in terms of the post-conditions of the sequences of operations, specified also in the object constraint language.

2.7. Data Flow Analysis

One concern in software testing is whether the program variables are appropriately created and used. Data flow analysis has been proposed to address this issue. For example, there will be an anomaly if a variable is referenced before it is assigned any value. A dynamic approach, known as dynamic data flow analysis, further uses the program instrumentation technique to insert probes to follow the actions on variables during program execution. It has been successfully applied to integration testing of conventional programs [27].

In recent years, data flow analysis has been extended to object-oriented programs [28, 29] and synchronization anomalies [30] based on the Java construct. Chen and Low [29] have also proposed a technique for C++. It is based on a special characteristic of the language such that the memory locations of individual variables referenced by a program can be retrieved and compared during execution.

Nevertheless, current work on data flow analysis of object-oriented programs has simply applied conventional techniques to the new paradigm. They do not make use of the specific features of object-oriented programs. For example, some of the anomalies due to the new paradigm may be overlooked. It will be interesting to see whether an object-oriented approach to data flow analysis can be developed for integration testing in the new paradigm.

3. Conclusion

We have presented an overview of research work on integration testing for object-oriented programs.

(*a*) The state-based approach uses interacting finite state machines to model an integrated system. The difficulty of the technique increases when the number of concurrent units increases. This is not consistent with the fact that dynamic instantiation of objects during program execution is an intrinsic part of object-oriented programming.

(*b*) The event-based paradigm uses relationships among events to harness the system and check for constraints violation. Heterogeneous treatment of the same type of event across different objects or different states may warrant further research.

(*c*) Integrated formal methods appear to provide a promising track. However, the complexity of the combined formal models may be a serious concern for people in the industry.

(*d*) Deterministic testing forces the synchronization to be executed in desirable orders so that a deterministic test oracle can be applied and checked with the deterministic result. Because of dynamic binding, the same synchronization may result in different binding effects at different points of input, thus causing confusion.

(*e*) For fault-based testing, the effectiveness of mutation rules for integration testing of object-oriented programs has yet to be studied.

(*f*) UML-based techniques conduct testing by constraining UML toolkits with a formal notation. The integration of formal and practical techniques is a promising area.

(*g*) Dynamic data flow testing helps identify anomalies of data actions by collecting information during program executions. Conventional probing techniques may not be adequate for languages that support Java-like reflection. More research in this area will be warranted.

A common characteristic is that most of the reviewed approaches simply extend the techniques used in the imperative programming paradigm. Very few of them are genuine object-oriented testing techniques. This is certainly an appropriate area for further research.

References

[1] H. Y. Chen, T. H. Tse, and T. Y. Chen. TACCLE: a methodology for object-oriented software testing at the class and cluster levels. *ACM Transactions on Software Engineering and Methodology*, 10(1):56–109, 2001.

[2] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, 1994.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *ACM International Symposium on Software Testing and Analysis (ISSTA '02)*. ACM Press, New York, 2002.

[4] H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen. In black and white: an integrated approach to class-level testing of object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 7(3):250–295, 1998.

[5] S. Lyer and S. Ramesh. Apportioning: a technique for efficient reachability analysis of concurrent object-oriented programs. *IEEE Transactions on Software Engineering*, 27(1):1037–1056, 2001.

[6] S. C. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 5(4):334–377, 1996.

[7] P. V. Koppol, R. H. Carver, and K.-C. Tai. Incremental integration testing of concurrent programs. *IEEE Transactions on Software Engineering*, 28(6):607–623, 2002.

[8] R. H. Carver and K.-C. Tai. Use of sequencing constraints for specification-based testing of concurrent programs. *IEEE Transactions on Software Engineering*, 24(6):471–490, 1998.

[9] J. A. Goguen and J. Meseguer. Unifying functional, object-oriented, and relational programming with logical semantics. In *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner (eds.), pages 417–477. MIT Press, Cambridge, Massachusetts, 1987.

[10] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. In *Proceedings of the 5th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '90), ACM SIGPLAN Notices*, 25(10):169–180, 1990.

[11] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, Boston, 2000.

[12] S. A. Schneider. *Concurrent and Real-time Systems: the CSP approach*. John Wiley, New York, 1999.

[13] K. H. Chang, S. S. Liao, S. B. Seidman, and R. Chapman. Testing object-oriented programs: from formal specification to test scenario generation. *Journal of Systems and Software*, 42:141–151, 1998.

[14] J. E. Hong and D. H. Bae. High-level Petri net for incremental analysis of object-oriented system requirements. *IEE Proceedings: Software*, 148(1):11–18, 2001.

[15] G. Smith and J. M. Derrick. Specification, refinement and verification of concurrent systems: an integration of Object-Z and CSP. *Formal Methods in System Design*, 18(3):249–284, 2001.

[16] D. L. Brening. Systematic testing of multithreaded Java programs. M. S. Thesis, MIT, Cambridge, Massachusetts, 1999.

[17] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[18] J. D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Analysis of object-oriented programs: efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 258–269. ACM Press, New York, 2002.

[19] H. S. Seo, I. S. Chung, B. M. Kim, and Y. R. Kwon. The design and implementation of automata-based testing environment for Java multi-thread programs. In *Proceedings of the 8th Asia Pacific Software Engineering Conference (APSEC '01)*, pages 221–228. IEEE Computer Society Press, Los Alamitos, California, 2001.

[20] G. H. Hwang, K.-C. Tai, and T. L. Huang. Reachability testing: an approach to testing concurrent software. *International Journal of Software Engineering and Knowledge Engineering*, 5(4):246–255, 1995.

[21] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An extended overview of the Mothra software testing environment. In *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis*, pages 142–151. IEEE Computer Society Press, New York, 1988.

[22] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface mutation: an approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, 2001.

[23] H. Yoon, B. Choi, and J. O. Jeon. Mutation-based inter-class testing. In *Proceedings of Asia Pacific Software Engineering Conference (APSEC '98)*, pages 174–181. IEEE Computer Society Press, Los Alamitos, California, 1998.

[24] S. W. Kim, J. A. Clark, and J. A. McDermid. Investigating the effectiveness of object-oriented testing strategies with the mutation method. *Software Testing, Verification and Reliability*, 11(4):207–225, 2001.

[25] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual*. Object Technology Series. Addison Wesley, Reading, Massachusetts, 1998.

[26] L. Briand and Y. Labiche. A UML-based approach to system testing. In *Proceedings of the 4th International Conference in Unified Modeling Language (UML 2001)*, volume 2185 of *Lecture Notes in Computer Science*, pages 194–208. Springer-Verlag, Berlin, Germany, 2001.

[27] F. T. Chan and T. Y. Chen. AIDA: a dynamic data flow anomaly detection system for Pascal programs. *Software: Practice and Experience*, 17(3):227–239, 1987.

[28] A. S. Boujarwah, K. Saleh, and J. Al-Dallal. Dynamic data flow analysis for Java programs. *Journal of Information and Software Technology*, 42(11):765–775, 2000.

[29] T. Y. Chen and C. K. Low. Error detection in C++ through dynamic data flow analysis. *Software: Concepts and Tools*, 18(1):1–13, 1997.

[30] K. Saleh, A. A. Boujarwah, and J. Al-Dallal. Anaomaly detection in concurrent Java programs using dynamic data flow analysis. *Information and Software Technology*, 43(15):973–981, 2001.