OConGraX – Automatically Generating Data-Flow Test Cases for Fault-Tolerant Systems

Paulo R.F. Nunes, Simone Hanazumi, and Ana C.V. de Melo

Department of Computer Science,
University of São Paulo, São Paulo, Brazil
prnunes@ime.usp.br, hanazumi@ime.usp.br, acvm@ime.usp.br

Abstract. The more complex to develop and manage systems the more software design faults increase, making fault-tolerant systems highly required. To ensure their quality, the normal and exceptional behaviors must be tested and/or verified. Software testing is still a difficult and costly software development task and a reasonable amount of effort has been employed to develop techniques for testing programs' normal behaviors. For the exceptional behavior, however, there is a lack of techniques and tools to effectively test it. To help in testing and analyzing fault-tolerant systems, we present in this paper a tool that provides an automatic generation of data-flow test cases for objects and exception-handling mechanisms of Java programs and data/control-flow graphs for program analysis.

1 Introduction

The more complex to develop and manage systems the more software design faults increase[1] and, today, a significant amount of code is dedicated to error detection and recovery[2]. Exception-handling mechanisms provide a clear separation of codes for error recovery and normal behavior, helping in decreasing code complexity and software design faults. Due to these benefits, exception-handling is recognized as a good approach to provide fault-tolerant software[1].

To ensure the quality of fault-tolerant systems, normal and exceptional behaviors must be tested. One promising approach is the data-flow testing, which focuses on data status[3], analyzing the life cycle of data to find out unexpected or anomalous behaviors. However, testing programs is still an expensive software development activity because it is aimed at dynamically analyzing the product. One of the main problems is related to the number of test cases necessary to cover the whole program, and test criteria [4,5,6,7,8,9] have been created with this aim. These criteria establish how much a program must be tested to achieve certain quality requirements. Although they reduce the test space, there is still a large set of test cases, making in practice the test activity neglected if a tool support is not provided to support automating test cases and suites.

To help in reducing the test activity cost and enhancing program analysis, this paper presents *OConGraX* (*O*bject *Con*trol-Flow *Gra*ph with e*X*ceptions), a tool that generates the data-flow testing requirements for objects and exceptions

M. Núñez et al. (Eds.): TESTCOM/FATES 2009, LNCS 5826, pp. 229–234, 2009.
© IFIP International Federation for Information Processing 2009

of Java programs. Using *OConGraX*, testers can concentrate on generating test suites for the corresponding data-flow test cases, instead of manually creating graphs and test coverage requirements, reducing software development cost.

In this paper, some concepts of data-flow techniques will be presented in Sect. 2. Section 3 presents the tool and its use for testing the normal and exceptional behaviors. Finally, Sect. 4 concludes presenting practical uses of the tool, its limitations and future works.

2 Data-Flow Testing and Fault-Tolerant Programs

Fault-tolerant programs must comprise the *normal behavior* code, in which errors are detected and the corresponding exceptions are raised; and the *exception handler* code[10], in which the exceptional behavior is treated resulting in the error recovery. To test them, a set of techniques that explore object-flow information was developed to guide test cases selection in the object-oriented paradigm[6,8,9]. The tool presented here is based on two coverage criteria: for normal the behavior, the criteria proposed by Chen and Kao[6]; and for the exceptional behavior, the criteria proposed by Sinha and Harrold[9,8]. Some concepts related to both criteria are briefly discussed here:

Object-definition: when the constructor of the object is invoked; the value of an attribute is explicitly assigned; or a method that initiates or update attributes of the object is invoked.

Object-usage: an object is used when one of its attributes is used in a predicate or a computation; one method that uses an attribute value is invoked; or an object is passed as parameter in a method invocation.

DU Pairs: all **du** (definition and use) pairs relating a definition and further use (without redefinition) of an object in the program control-flow.

Exception: *Object*: an exception class instance denoted as $eobj_i$, where i corresponds to the code-line in which it is instantiated. *Variable*: a variable of type exception. *Temporary Variable*: an exception related to a *throw* command – represented by $evar_i$, where i corresponds to the *throw* code-line. *Object activation/deactivation*: an exception object is activated when it is raised with a *throw* command and deactivated when it is treated or a *null* value is given (an active object is represented by $evar_{active}$, and it is unique at each execution instant).

Exception-definition: a value is assigned to an exception variable or to an exception variable in *catch*; or a temporary variable is used in *throw*.

Exception-usage: an exception variable value is used; a variable value is used in *catch*; or a temporary exception variable is used in *throw*.

E-DU Pairs: a value is assigned to an exception variable that is further used in a *throw*; a parameter has its value accessed in a *catch* block; an exception variable is activated and deactivated (treated).

Object coverage criterion: *all-du-pairs*: all **du** pairs are tested.

Exception coverage criteria: *all-throw*: all *throw* commands are tested; *all-catch*: all *catch* commands are tested; *all-e-defs*: all exception definitions, associated to at least one use, is tested; *all-e-use*: all **e-du** pairs are tested; *all-e-act*: all activated exception, related to at least one deactivation, is tested; *all-e-deact*: all pairs of exception activation-deactivation (e-ad) pairs are tested.

To satisfy those criteria[6,9,8], test cases must be created to cover all object and exception **du** and act-deact-pairs. To identify the actual **du** pairs, one must first identify all **du** of objects and exceptions and create the object and exception control-flow graphs (OCFG). Building the OCFG for objects and exceptions is very error-prone and cannot be left to users. The forthcoming section presents a tool to automate these steps.

3 *OConGraX* – Test Cases for the Normal and Exceptional Behaviors

OConGraX has been created to help in testing (data-flow) and analyzing Java programs. The tool reads a Java Project and provides the following basic services:

- Definitions, uses and def-use pairs of objects: the set of test cases based on the criterion of **all definition-use** (**du**) pairs of objects are generated;
- Definitions, uses and def-use pairs of exceptions: the set of test cases based on the criterion of **all definition-use** (**e-du**) pairs of exceptions are generated;
- OCFG with the additional information on exception-handling;
- Graphs and test cases exported as images or XML files.

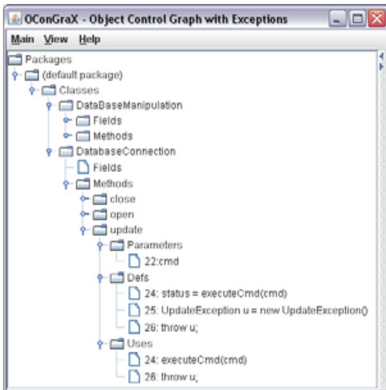
OConGraX can be used to automatically generate test cases for normal and exceptional behaviors of Java systems. This section shows the use of the tool to this purpose. First, an example is presented (Fig. 1) followed by the data-flow

```
 1: public class DataBaseManipulation {
 2:   DatabaseConnection dbConn =
               new DatabaseConnection();
 3:   String sqlCmd = "";
 4:   Array row;
 5:
 6:   void dbUpdateOperation () {
 7:     dbConn.open();
 8:     try{
 9:       row = dbConn.select(sqlCmd);
10:       sqlCmd = updateFields(row);
11:       dbConn.update(sqlCmd);
12:     }
13:     catch(UpdateException ue) {
14:       showMessage(ue);
15:     }
16:     finally {
17:       dbConn.close();
18:     }
19:   }
20: }
21: public class DatabaseConnection {
22:   void update(String cmd) throws
                      UpdateException {
23:     int status;
24:     if ((status = executeCmd(cmd))==0) {
25:       UpdateException u = new
                      UpdateException();
26:       throw u;
27:     }
28:   }
29:   void open() {//open DB connection}
30:   void close() {//close DB connection}
31: }
```

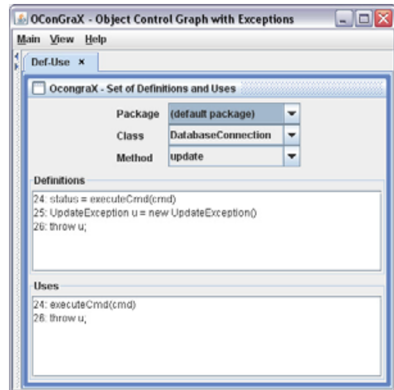
Fig. 1. Example code

test cases generated for the normal and exceptional behaviors. The example was adapted from the one showed in [9] and illustrates a simple database connection and update operations. The operations are invoked in `DataBaseManipulation` class and they are implemented in `DataBaseConnection` class.

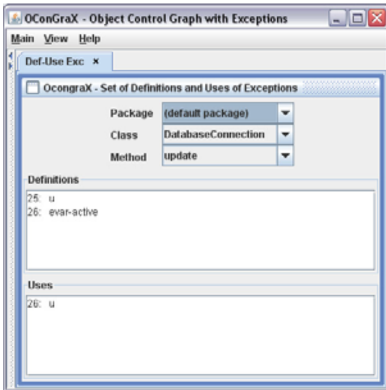
In *OConGraX*, once an input is chosen, a tree with all information about the selected project can be accessed as shown in Fig. 2(a). To obtain the definitions and uses of objects (and exceptions), one can select from the "View" menu the option "Def-Use". A tab will come along on the right panel in which the Package, Class and Method can be selected and the code lines where a definition or a use occur will be presented. Figure 2(b) shows this for the example when **package:** `default package`, **class:** `DatabaseConnection` and **method:** `update` are selected. To view only the definitions and uses of exceptions, one can select the option "Def-Use Exc" – Fig. 2(c). The **du** (or **e-du**) pairs are obtained by selecting the "Def-Use Pairs" option together with the class and object (or exception) – they are shown as `(<def.`



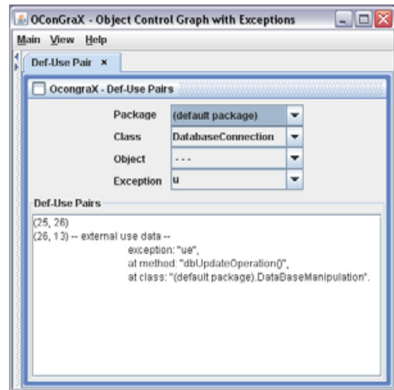
(a) Software source information tree



(b) Definitions and uses view



(c) Exception definitions and uses view



(d) Exception du pairs view

Fig. 2. Tool features related to definitions and uses view

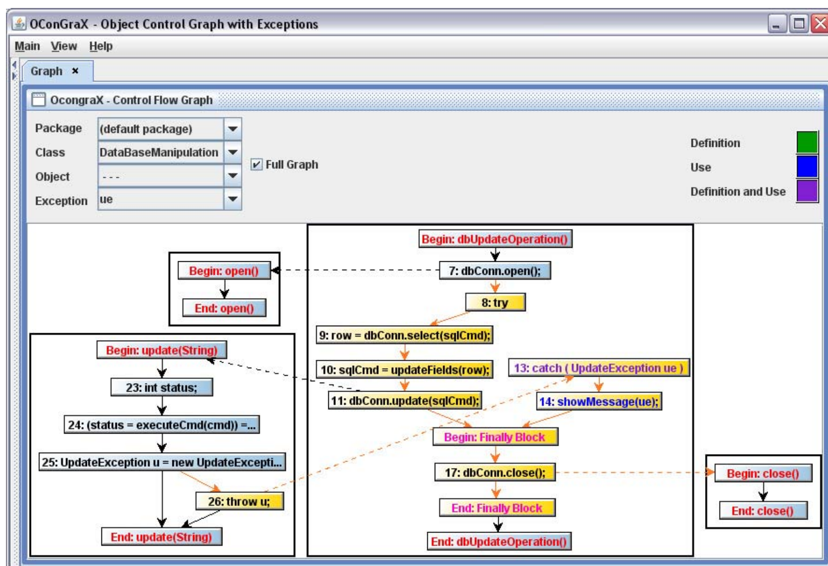


Fig. 3. Graph View

code-line>, <use code-line>). Figure 2(d) shows the **e-du** pairs for the example when **class:** `DataBaseManipulation` and **exception:** `ue` are selected. The OCFG with exceptions can also be obtained by selecting the "Graph" option and then the Package, Class and Object/Exception. The option "Full Graph" shows the whole graph, otherwise only parts of the graph related to the selected object will appear. Figure 3 shows the graph obtained when we select **package:** `default package`, **class:** `DataBaseManipulation` and **exception:** `ue` with the "Full Graph" option selected.

4 Concluding Remarks

Despite being widely used, the exception-handling mechanisms are mainly defined at program level, instead of at the design level. As a result, assuring fault-tolerant systems functionalities relies on testing and verification at program level. Testing a program depends very much on tools support; otherwise this activity could be neglected in the development process to reduce costs.

The data-flow testing criteria are based on code coverage and can be defined with static analysis. To automate the test cases generation, this paper presented *OConGraX*, a tool that can detect definition and use of objects and exceptions; generate du-pairs of objects and exceptions; generate the OCFG (for objects and exceptions); and export information about **du** and **e-du** pairs as XML files and graphs as images files. To actually test programs, the test suites still need to be created. This feature is not yet provided by the tool and we are now investigating the viability of integrating it with tools to generate test data. *OConGraX* has

been developed to automate test cases for objects and exceptions under data-flow testing techniques, but can be used to help in understanding programs and be integrated with verification tools to take advantages of the V&V approach. In [11], we suggested an approach, and a corresponding tool support, to integrate and automate testing and verification activities for fault-tolerant systems.

Although *OConGraX* presents useful features, many other features remain to be developed: extracting code measures automatically, such as number of object usages; introducing other code coverage criteria, such as the ones based on decisions; and generating data test. Besides theses, developing new parsers for other Object-Oriented programming languages can make all the *OConGraX* features available for other languages – more widely applied.

Acknowledgments. This project has been co-funded by the National Council for Scientific and Technological Development (CNPq - Brazil), the State of São Paulo Research Foundation (FAPESP) and the Ministry of Education Research Agency (CAPES - Brazil). The author Ana C. V. de Melo also thanks the Oxford University Computing Laboratory for providing research facilities during her stay on sabbatical leave at the University of Oxford.

References

1. Lee, P.A., Anderson, T.: Fault Tolerance: Principles and Practice. Springer, Secaucus (1990)
2. Randell, B.: The evolution of the recovery block concept. In: Lyu (ed.) Software Fault Tolerance, pp. 1–21 (1995)
3. Badlaney, J., Ghatol, R., Jadhwani, R.: An introduction to data-flow testing. Technical Report (2006) 22, http://www.csc.ncsu.edu/research/tech/reports.php (Last access Feburary 2009)
4. Weyuker, E.J.: The evaluation of program-based software test data adequacy criteria. Commun. ACM 31(6), 668–675 (1988)
5. McGregor, J.D., Korson, T.D.: Integrated object-oriented testing and development processes. Commun. ACM 37(9), 59–77 (1994)
6. Chen, M.H., Kao, H.M.: Testing object-oriented programs – an integrated approach. In: Proceedings of ISSRE 1999, p. 73. IEEE Computer Society, Los Alamitos (1999)
7. Kung, D., Suchak, N., Hsia, P., Toyoshima, Y., Chen, C.: Object state testing for object-oriented programs. In: Proceedings of COMPSAC 1995, p. 232. IEEE Computer Society, Los Alamitos (1995)
8. Sinha, S., Harrold, M.J.: Criteria for testing exception-handling constructs in Java programs. In: Proceedings of ICSM 1999, p. 265. IEEE Computer Society, Los Alamitos (1999)
9. Sinha, S., Harrold, M.J.: Analysis of programs with exception-handling constructs. In: Proceedings of ICSM 1998, p. 348. IEEE Computer Society, Los Alamitos (1998)
10. Cristian, F.: Exception handling and software fault tolerance. In: FTCS-25: Highlights from Twenty-Five Years, p. 120. IEEE Computer Society, Los Alamitos (1995)
11. Xavier, K.S., Hanazumi, S., de Melo, A.C.V.: Using formal verification to reduce test space of fault-tolerant programs. In: Proceedings of SEFM 2008, pp. 181–190. IEEE Computer Society, Los Alamitos (2008)