FORMAL SPECIFICATION-BASED MONITORING, REGRESSION TESTING AND ASPECTS

LIANG HUI

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2007

Acknowledgement

First and foremost, I would like to express my sincere gratitude and respect to my supervisor, Dr. DONG Jin Song, for supporting me with guidance, encouragement, inspiration, patience and everything else I needed as a student throughout my Ph.D. study. It would not be possible to complete this thesis without his constant support. Dr. Dong has provided me with lots of technical and professional advice. This makes working with him a true privilege; and there is no doubt that I will benefit from his instructions throughout my career.

I owe many thanks to my co-supervisor, Dr. SUN Jing, for his constructive and supportive feedbacks to the work presented in this thesis, his suggestions on all aspects of research and his continuous encouragement.

I am indebted to Dr. Roger Duke, Dr. Rudolph E. Seviora and Dr. Dines Bjørner for many helpful discussions and their insightful feedbacks on the work presented in this thesis. Their perspectives enlarged my view and helped improve the work.

I am deeply grateful to Dr. Khoo Siau Cheng and Dr. Bimlesh Wadha for the valuable comments and constructive suggestions for the improvement of this thesis.

I am also grateful to the external examiner and many anonymous reviewers who have reviewed this thesis and my papers, on which this thesis is based, and provided valuable feedbacks and comments that have contributed to the clarification and improvement of many of the ideas presented in this thesis.

I would like to thank all the past and present members of our research group. Despite the span of research topics that we have been exploring, there have always been great feedbacks and supports from all of them. I want to thank all the members in the Software Engineering Lab for providing a nice, friendly and quite environment.

My gratitude also goes to the National University of Singapore for providing financial supports for my Ph.D. study. The School of Computing provided me with grants for presenting papers in several conferences overseas. For all of these, I am very grateful.

I would also like to take this opportunity to thank all the administrative and technical support staffs at the School of Computing. It is only because of them that everything - from fixing problems with the network connection to organizing a conference travel - seemed so easy.

A large group of friends have also made an invisible but valuable contribution to this thesis by being there. The weekly badminton games with some of them raised the state of my mind and body.

Last but not least, I am sincerely grateful to my family. My parents and younger brother are always there for me whenever I need encouragements and supports.

Contents

1 Introduction **1**

1.1 Motivation and Goals 1

1.2 Thesis Outline 6

1.2.1 Chapter 2 6

1.2.2 Chapter 3 6

1.2.3 Chapter 4 7

1.2.4 Chapter 5 7

1.2.5 Chapter 6 7

1.2.6 Chapter 7 8

1.2.7 Chapter 8 9

1.3 Publications 9

2 Background and Related Work **11**

2.1 Formal Specification Languages 12

2.1.1 Z 13

2.1.2 TCOZ 15

2.2 Software Monitoring 17

2.3 Regression Testing 20

2.4 AOSD and Formal Methods 23

3 Formal Specification-based Software Monitoring **27**

3.1 Introduction 28

3.2 Specification Animation 30

3.3 Formal Specification-based Software Monitoring 31

3.3.1 Overview of the Technique 32

3.3.2 A Prototype 33

3.3.3 Technical Challenges 36

3.4 Discussion 44

3.4.1 Merits 44

3.4.2 Limitations 46

3.4.3 Impacts on Software Testing 47

3.5 Conclusion 49

4 Monitoring System Case Studies **51**

4.1 Introduction 52

4.2 A Railway Control System 52

4.3 A Robotic Assembly System 57

4.4 Conclusion 64

5 AOP-aided Software Evolution **65**

5.1 Introduction 66

5.2 AOP and AspectJ 68

5.2.1 AOP 68

5.2.2 AspectJ 69

5.3 AOP-aided Software Evolution 70

5.3.1 Determine Differences 71

5.3.2 Construct Aspects 73

5.3.3 Weave Constructed Aspect with Original Class 74

5.4 Monitor Aspect-oriented Programs 75

5.5 A Case Study 76

5.6 Conclusion 81

6 Formal Specification-based Regression Test Suite Construction 83

6.1 Introduction 84

6.2 Classes Specified in TCOZ Notation 86

6.3 Representation for Classes Specified in TCOZ 88

6.3.1 Testchart 89

6.3.2 Coverage Criteria 91

6.4 Regression Test Suite Construction 93

6.4.1 Overview of the Technique 94

6.4.2 Modifications to TCOZ Specification 94

6.4.3 Impacts of Modifications on Test Cases 97

6.4.4 Regression Test Selection Algorithm 98

6.4.5 A Case Study 100

6.5 TCOZ-based Regression Test Suite Construction System 103

6.6 Conclusion 105

7 Formal Specification Notation for AOSD **107**

7.1 Introduction 108

7.2 Overview of a Simple Telephone System 110

7.3 AspecTCOZ - an Extension of TCOZ 112

7.3.1 Join Point 113

7.3.2 Pointcut 114

7.3.3 Advice 118

7.3.4 Inter-type Declaration 121

7.3.5 Aspect 123

7.4 Formal Specification-based Aspect Conflict Detection 124

7.5 Conclusion 127

8 Conclusion **129**

8.1 Main Contributions of the Thesis 130

8.2 Future Work Directions 132

8.2.1 Further Development of Monitoring Technique 132

8.2.2 Formal Methods for AOSD 133

A Specification of Railway Control System in Z Notation **155**

B Implementation of Railway Control System in Java **157**

C Specification of Robotic Assembly System in Z Notation **159**

D Implementation of Robotic Assembly System in Java **163**

Summary

With the sound mathematical basis and well-defined semantics and syntax of formal languages, the formal specification of a software system provides deep insight into and precise understanding of system requirements. It also provides a powerful basis for software verification and validation. This thesis explores parts of the potential of formal specifications in contributing to high quality software.

A formal specification-based software monitoring approach is proposed in this thesis. Based on formal specification animation and program debugging, the proposed software monitoring approach dynamically and continuously checks the conformance of concrete implementations to formal specifications, explicitly recognizes undesirable behaviors in the target system, and responds appropriately in a timely manner as the target system runs.

Frequently, the formal specification of a software system has to change according to the changes of system requirements. Correspondingly, the implementation of the software system has to be changed in order to keep conformance with the formal specification. Taking advantage of aspect-oriented programming technique, we propose an approach for handling the evolution of core classes in object-oriented programs when the formal specification of a system has changed.

After the software has evolved according to the changes of the formal specification, regression testing has to be performed to ensure that the changed parts of the software behave as intended and that the unchanged parts have not been adversely affected by the modifications. To reduce the cost of regression test and deal with the conditions that cannot be handled by code-based regression test selection techniques, a formal specification-based regression test suite construction approach is proposed in this thesis.

Aspect-oriented software development (AOSD) is a new promising methodology. However, validation techniques for aspect-oriented programs are still far from sufficient. With the expectation that formal methods could be applied to aspect-oriented programs in the future, we extend the integrated formal notation Timed Communicating Object-Z (TCOZ) with the mechanisms for formally specifying those aspect-oriented constructs, providing a starting point for future research work on the development of formal methods for aspect-oriented software development.

List of Tables

7.1 Formal notations for join point model of AspectJ 114

7.2 Formal notations for pointcut designators of AspectJ 115

List of Figures

2.1 *Queue* in Z notation 14

2.2 *TimedQueue* in TCOZ notation 16

3.1 Formal specification-based software monitoring system 29

3.2 Debugging mode 34

3.3 Running mode 35

3.4 *Queue* in Z notation 37

3.5 Matching between concrete implementation and formal specification 38

3.6 Class *Queue* in Java 40

3.7 Inconformity occurs 41

3.8 Handle nondeterminism. 43

4.1 Track line signalling 53

4.2 Monitor a railway control system 55

4.3 Robotic assembly system 58

4.4 Monitor a robotic assembly system 62

5.1 Difference determination 72

5.2 Aspect construction 73

5.3 *TimedQueue* in TCOZ notation 77

5.4 Modified *TimedQueue* in TCOZ notation 78

5.5 Aspect for ineligible element rule 79

5.6 Monitoring aspect-oriented program 80

5.7 Correct advice 80

6.1 *TimedQueue* in TCOZ notation 90

6.2 Testchart for *TimedQueue* 92

6.3 Formal specification-based regression test selection algorithm 99

6.4 Modified *TimedQueue* 101

6.5 Testchart for modified *TimedQueue* 102

6.6 TCOZ-based regression test suite construction system: TcozRts . . 104

7.1 The *Connection* class 111

7.2 The *Customer* class 112

7.3 An *after* advice 120

7.4 *TimingBilling* aspect in AspecTCOZ notation 124

Chapter 1

Introduction

1.1 Motivation and Goals

Nowadays, we are going into a ubiquitous computing world where software permeates many aspects of our society, such as transportation, commerce and so on. Therefore, software quality is becoming critically important to the whole world.

With the sound mathematical basis and well-defined semantics and syntax of formal languages, the formal specification of a software system provides a powerful basis for software verification and validation. In this thesis, potentials of formal specifications in contributing to high quality softwares will be explored.

To improve software quality, many researchers have concentrated on the methods of analysis and validation for software systems, especially for the softwares deployed in safety critical areas such as avionics and medicine. Lots of progress has been

achieved in the area of formal verification. However, complete formal verification is still widely considered to be prohibitively expensive to apply to nontrivial real-life systems. The growth of software size and complexity always exceeds the advances in verification technology. Moreover, the results of verification apply to formal models of these systems, but not to the system implementations. This means that the reliability and correctness of a particular implementation of a system can not be assured by the formal verification of the system's model.

Alternatively, software engineers resort to testing to verify the conformance of implementation with design. Compared with formal verification, testing is a less rigorous method for validating the correctness of software systems, but it is feasible to test large and complex systems in terms of cost. However, testing is usually incompetent to provide guarantees about the correctness of a particular implementation on all possible input sequences. Hampered by the need to independently compute and verify the expected outputs for each test case, testing is usually conducted with a reasonable pre-determined subset of all possible input sequences.

Consequently, the correctness of a system can not be guaranteed by the two validation methods that have been discussed above.

With the capabilities of detecting, diagnosing and recovering from software faults, software monitoring provides additional defense against software failure. It can be used as a complement to formal verification and software testing so that higher reliability of software systems will be achieved.

Therefore, the first goal of this thesis is to propose a formal specification-based

software monitoring approach, which can not only dynamically and continuously monitor the behaviors observed in the target system but also explicitly recognize undesirable behaviors in the target system with respect to the formal specifications of the system. With the proposed formal specification-based software monitoring technique, we aim to provide an approach that can be used as a complementary technique to formal verification, and can support software testing in effective test execution and automatically checking whether actual output of the program under test is equivalent to the expected output.

Oftentimes, the specification of a software system has to be changed because new requirements emerge or the running environment changes. Whenever the specification of a software system has changed, the corresponding modifications have to be made to the implementation. As a promising methodology, aspect-oriented programming (AOP) [53, 64, 100] provides a model to modify a software system after it has been released and installed, which greatly eases the maintenance and evolution of software systems.

The second goal of this thesis is to investigate how AOP techniques can contribute to the evolution of core classes in a system that is implemented in object-oriented programming language, when the formal specification of the system has changed. Furthermore, AOP not only brings a unique set of benefits, but also introduces a set of challenges, such as new problems with respect to the verification and testing of systems developed with AOP. By far, validation techniques for aspect-oriented programs are still far behind expectation. Because the expressive power that aspects

unleash heightens the potential for insidious errors, finding cost-effective validation techniques that address aspect-oriented programs is especially important to aspect-oriented software development. In order to validate the program resulting from AOP-aided evolution, we try to extend the formal specification-based monitoring approach that we have proposed so that it can work with aspect-oriented programs. By doing this, we provide an approach for aspect-oriented program validation.

After the software has evolved according to the changes of the specification or requirement of the software system, regression testing has to be performed to provide the confidence that the changed parts of the software system behave as intended and that the unchanged parts have not been adversely affected by the modifications. Most of the existing regression test selection techniques [9, 19, 59, 84, 106, 110, 48, 85, 109, 37] are strictly code-based. They select test cases for the regression testing, from the original test suites, only using the information gathered by code analysis. Thereby, they cannot deal with the conditions where the specifications of the software system have been changed, but the code modifications that are necessary to implement the changed specifications have not been made.

Therefore, the third goal of this thesis is to propose a formal specification-based regression test suite construction technique, which can serve as a complement to the existing code-based regression testing selection techniques, so that more effective and more comprehensive software regression testing can be achieved.

Aspect-oriented software development(AOSD) [33] is an emerging technology that

supports the encapsulation and modularization of concerns which crosscut the primary decomposition of a software system. The research in AOSD has achieved a lot in the design and implementation of aspect-oriented programming languages. However, the techniques for validating aspect-oriented programs are still far from sufficient. Meanwhile, existing formal methods can not be applied to AOSD directly because new concepts and constructs, such as pointcut, advice, inter-type declaration and aspect, are introduced to aspect-oriented programs.

With the expectation that the existing formal methods could be extended and applied to aspect-oriented programs, the fourth goal of this thesis is to extend the integrated formal notation TCOZ (Timed Communicating Object-Z) [66, 70] with the mechanisms for formally specifying those aspect-oriented constructs, to provide a starting point for future research work on the development of formal methods for aspect-oriented software development.

Meanwhile, in AOP, multiple aspects are allowed to be superimposed on the same join point. Consequently, undesired or incorrect behavior may emerge due to unexpected conflicts between aspects. The development of effective mechanisms for detecting those conflicts between aspects is critical to the maturity of AOP. Furthermore, early detection of those conflicts will make it possible to reduce the development cost while promising a high quality software system. Therefore, as part of the fourth goal, we try to propose an approach for the early detection of conflicts between aspects, based on the formal specification of an aspect-oriented software system.

1.2 Thesis Outline

This section presents an overview of the structure of this thesis.

1.2.1 Chapter 2

Chapter 2 introduces background information on formal specification languages and software development/maintenance techniques covered by this thesis, and surveys related work.

1.2.2 Chapter 3

Chapter 3 presents a formal specification-based monitoring technique. In the proposed monitoring technique, the valuable information about expected dynamic behaviors of the target system is extracted through animating the formal specification of the system. Meanwhile, the information about actual dynamic behaviors of concrete implementations of the target system is obtained through program debugging. Base on the information obtained from both sides, the judgement on the conformance of the concrete implementation with the formal specification is timely made while the target system is running.

1.2.3 Chapter 4

Chapter 4 demonstrates the application of the formal specification-based monitoring technique proposed in Chapter 3 with case studies.

1.2.4 Chapter 5

Chapter 5 investigates how AOP (aspect-oriented programming) [53, 64] techniques can contribute to the evolution of core classes in object-oriented programs when the formal specification of the system has changed. As a result, an AOP-aided software evolution approach is proposed. First, the old and new versions of the formal specification of a software system are compared to identify the differences. Second, aspects are constructed to achieve the expected modifications. After weaving the constructed aspects with original classes, the required evolution will be accomplished. Furthermore, the formal specification-based monitoring system presented in Chapter 3 is extended to validate the modified software resulting from the AOP-aided evolution.

1.2.5 Chapter 6

Chapter 6 presents a formal specification-based technique for the construction of regression test suite. The proposed technique addresses the *regression test selection problem* and *test suite augmentation problem* for the regression testing of classes specified in TCOZ notation. It constructs control flow representations for

the classes; then compares the original versions of the representation and the specification with their respective modified version to figure out the modifications made to the specification; and sequentially uses the information about the modifications made to the specification to identify the obsolete test cases, to select the test cases related to the changed specifications from the original test suite, and to guide the generation of new test cases for regression testing.

1.2.6 Chapter 7

Chapter 7 presents a formal specification notation for aspect-oriented software development. The class schema in TCOZ notation is an eligible candidate for specifying an aspect formally because *aspect* is the unit of modularity, encapsulation, and abstraction in AOP, just in the same way as *class* in OOP. Meanwhile, the strength of TCSP in modeling process control and real-time interactions, which is preserved in TCOZ, provides a great mechanism for specifying the temporal order between pointcut and advice. Therefore, we try to extend TCOZ with the mechanisms for formally specifying the constructs of join point, pointcut, advice, and inter-type introduction. Consequently, AspecTCOZ, as an aspect-orientated extension of TCOZ, provides a starting point for future research work on the development of formal methods for aspect-oriented software development. Furthermore, Chapter 7 presents a formal specification-based approach for the early detection of conflicts between aspects when multiple aspects are superimposed on the same joint point.

1.2.7 Chapter 8

Chapter 8 concludes this thesis with a summary of the main contributions and discussions about future work directions.

1.3 Publications

Most of the work presented in this thesis has been published in the proceedings of international conferences.

The work on the formal specification-based monitoring (Chapter 3) has been published in *The 11th IEEE International Conference on Engineering of Complex Computer Systems* (ICECCS'06, August 2006, Stanford University) [62]. The work on AOP-aided approach for software evolution and application of formal specification-based monitoring technique to aspect-oriented programs (Chapter 5) has been published in *The Nineteenth International Conference on Software Engineering and Knowledge Engineering* (SEKE'07, July 2007, Boston) [61]. The work on formal specification-based regression test suite construction (Chapter 6) has been published at *The 10th IEEE International Conference on Engineering of Complex Computer Systems* (ICECCS'05, June 2005, Shanghai) [60]. The work on formal specification notation for aspect-oriented software development and the formal specification-based approach for aspect conflicts detection (Chapter 7) has been published in *The Nineteenth International Conference on Software Engineering and Knowledge Engineering* (SEKE'07, July 2007, Boston) [63].

Chapter 2

Background and Related Work

This chapter presents the background information on the formal specification languages and software development/maintenance techniques covered by this thesis, and discusses how our work relates to other research.

2.1 Formal Specification Languages

Formal methods are mathematically precise notations, tools, and techniques used for the development of software systems. The use of formal methods can result in software systems with fewer faults. The cornerstone of formal methods is the specification of the software system which is expressed in a mathematically precise formal notation. The well-defined semantics and syntax of formal specification languages make it possible for the formal specification to be validated informally by expert inspection, or formally using tool-assisted theorem proving techniques. The resulting specification is usually more precise, unambiguous, and complete than an informal natural-language specification.

Many formal specification languages have been proposed to depict various aspects of software systems from different perspectives. For example, VDM [13], Z [93], Object-Z [92], and B [5] are state-oriented formalisms; ACT1 [32], CLEAR [16], OBJ [34], and Larch [36] are algebraic formalisms and CSP [46]; TCSP [89], CCS [77], and LOTOS [15] are process-oriented formalisms. Furthermore, the design of complex systems requires powerful mechanisms for modeling data, state, communication, and real-time behavior; and the mechanisms for structuring and decomposing systems as well. Therefore, efforts have been made to develop integrated formal specification languages, which combine different formalisms to capture static and dynamic system properties in a highly structured way. The achievements in this research area include TCOZ(Timed Communicating Object-Z) [66, 70], SOFL(Structured Object-oriented Formal Language) [65] and so on.

This section introduces the two formal specification languages that will be used in this dissertation: Z and TCOZ.

2.1.1 Z

The Z specification language has been a widely accepted formal language for specifying the behaviours of software and hardware systems. Based on set theory and first order predicate logic, Z is a model oriented specification language. It models a system by describing its states and the ways in which the states can be changed. This modeling style makes Z not only a good match to imperative, procedural programming languages but also a natural fit to object-oriented programming [49, 93]. Actually, the Z specification language includes two parts: the mathematical language and the schema language [112]. The specification written in Z typically includes a number of state and operation schema definitions. A state schema encapsulates variable declarations and related predicates (invariants). The system state is determined by values taken by variables subject to restrictions imposed by state invariants. An operation schema defines the relationship between the 'before' and 'after' states corresponding to one or more state schemas. The schema language can be used to structure and compose the formal descriptions of more complex operations by using schema calculus, such as sequential composition ' $\fatsemi$ ', conjunction ' $\wedge$ ', disjunction ' $\vee$ ' implication ' $\Rightarrow$ ', negation ' $\neg$ ' and pipe ' $\gg$ '. Detailed information about the syntax and semantics of Z notation can be found in [93, 112].

$size == 10$

Queue

$items : \operatorname{seq} \mathbb{N}$

$\#items \leq size$

InitQueue

$Queue'$

$items' = \langle\rangle$

Enqueue

$\Delta Queue$
$item? : \mathbb{N}$

$\#items < size$
$items' = items \frown \langle item? \rangle$

Dequeue

$\Delta Queue$
$item! : \mathbb{N}$

$\#items > 0$
$item! = head\ items$
$items' = tail\ items$

DupOneinTail

$\Delta Queue$
$item! : \mathbb{N}$

$\#items > 1 \land \#items < size$
$item! \in \operatorname{ran} items \land item! \neq head\ items$
$items' = items \frown \langle item! \rangle$

Figure 2.1: *Queue* in Z notation

As an example, the Z specification shown in Figure 3.4 describes a queue which is a First In First Out(FIFO) queue in nature, but with the addition of the *DupOneinTail* operation. The *DupOneinTail* schema describes an operation that selects an item, which is not the first element, from the queue randomly and adds it to the end of the queue. It is obvious that *DupOneinTail* is a nondeterministic operation because there will be more one possible results for the execution of it when there are more than two items in the queue.

2.1.2 TCOZ

Timed Communicating Object-Z (TCOZ) [66, 70] is an integration and extension of the formal modeling notations: Object-Z [17, 30, 92] and Timed CSP [89, 90]. It is built on Object-Z's strengths in modeling complex data and algorithms, and Timed CSP's strengths in modeling process control and real-time interactions. The essence of the integration is the unification of the concepts of type, class, and process and the unification of Object-Z operation specification schemas with terminating CSP processes.

Besides, in TCOZ, the CSP channel plays an independent first-class role. This allows the communications and control topology of a network of objects to be designed orthogonally to their class structure. Complementary to the synchronizing CSP channel mechanism, two continuous (asynchronous) interface mechanisms: sensor and actuator, which are inspired by the process control theory, are introduced in TCOZ [69]. The sensor provides a sampling channel linked to a global analogue variable. The actuator provides a local-variable linked to a global analogue variable. Besides, the syntactic structure of the CSP synchronization operator is convenient only in the case of pipe-line like communication topologies. Expressing more complex communication topologies generally results in unacceptably complicated expressions. Therefore, a graph-based approach is adopted in TCOZ to represent the network topology [67]

Consequently, TCOZ provides a timed, multi-threaded object modeling notation for the design of complex systems. Detailed introduction to TCOZ and its Timed

$[MSG]$ [messages]

TimedQueue

$items : \text{seq}\, MSG$
$in, out : \textbf{chan}$
$lost : \text{seq}\, MSG\ \textbf{actuator}$
$T_o : \mathbb{N}$

INIT
$items = lost = \langle\,\rangle$

RecLost
$\Delta(lost)$

$items \neq \langle\,\rangle \Rightarrow lost' = \langle head(items)\rangle^\frown lost$
$items = \langle\,\rangle \Rightarrow lost' = lost$

Add
$\Delta(items)$
$i? : MSG$

$items' = items \frown \langle i?\rangle$

Del
$\Delta(items)$
$i! : MSG$

$items \neq \langle\,\rangle \Rightarrow items = \langle i!\rangle^\frown items'$
$items = \langle\,\rangle \Rightarrow items' = \langle\,\rangle$

$Join \mathrel{\widehat=} [i : MSG] \bullet in?i \rightarrow Add$
$Leave \mathrel{\widehat=} [items \neq \langle\,\rangle] \bullet out!head(items) \rightarrow Del$
$\textsc{Main} \mathrel{\widehat=} \mu\, Q \bullet (Join \,\Box\, Leave) \triangleright \{T_o\}\ (RecLost;\ Del);\ Q$

Figure 2.2: *TimedQueue* in TCOZ notation

CSP and Object-Z features may be found in [66, 70]. The formal semantics of TCOZ is documented in [68].

As an example, the TCOZ specification shown in Figure 2.2 describes a class of a simple timed message queue system. The *Timed Message Queue System* can receive a new message (of type $[MSG]$) through an input channel '*in*' or remove a message and send it through an output channel '*out*'. If there is no interaction within a certain time 'T_o', a message will be lost from the current (*items*) list and stored in an asynchronous actuator list (*lost*) so that other objects (un-specified) with

a sensor '*lost*' can read it at any time. The messages in the queue are removed in first-in-first-out manner. Note that the operations *Join, Leave* and *Main* are defined in terms of CSP processes while the operations *Add, Del* and *RecLost* are defined in form of the operation schemas. The state variables *in* and *out* are of type **chan**, and they will serve as the channel that connect the queue system and the environment.

2.2 Software Monitoring

Runtime software monitoring has been used for profiling, performance analysis, software optimization as well as for the purpose of detection, diagnosis, and recovery from software faults. It provides evidence that program behavior complies or does not comply with specified requirements during program execution. While other verification techniques, such as testing, model checking, and theorem proving, aim to ensure universal correctness of programs, the intention of runtime software monitoring is to determine whether the current execution preserves specified properties. Thus, software monitoring can be used to provide additional defense against catastrophic failure and to support test by exposing state information. The increasing complexity and ubiquitous nature of software systems and the limitation and inadequacy of current formal verification and software testing techniques have inspired renewed interest in the field of software monitoring [27].

Techniques and tools have been proposed for runtime software monitoring for tra-

ditional softwares. Java PathExplorer (JPaX) [38, 39, 40] is a runtime monitoring technique developed for sequential and concurrent Java programs. It facilitates logic-based monitoring and error pattern analysis. Formal requirement specifications are written in a linear temporal logic or in the algebraic specification language Maude [23, 22]. JPaX instruments Java byte code to transmit a stream of relevant events to the observation module that performs two kinds of analysis: logic-based monitoring (checking events against high-level requirements specification) and error pattern analysis (searching for low-level programming errors). Maude's rewriting engine is used to compare the execution trace to the specifications.

Monitoring and Checking (MaC) [54, 55, 56] provides a framework for runtime monitoring of real-time systems written in Java. In Mac, a monitoring script is used to monitor objects and methods; a filter maintains a table that contains names of monitored variables and address of corresponding objects, acting as an observer that communicates the information that is to be checked by the runtime monitor. Monitoring points are inserted automatically since the monitoring script specifies which information needs to be extracted. The event handler can emit a signal, or steer the program based on violation of specified conditions and actions which are provide by the users.

Java with Assertions (Jass) [11] is a general-purpose monitoring approach that is implemented for sequential, concurrent, and reactive systems written in Java. A precompiler translates annotations to programs written in Java into pure Java code. Compliance with the specified annotation is dynamically tested during run-

time. Assertions extend the design by contract approach that allows specification of assertions in the form of method pre and postconditions, class invariants, loop invariants, and additional checkers to be inserted at any part of the program code.

ProTest[88] is an automatic test environment for B specifications. After generating a set of test cases, ProTest simultaneously performs animation of the B machine and the execution of the corresponding implementation in Java, and assigns verdicts on the test results. This is kind of similar to our formal specification-based monitoring technique. However, in ProTest, the relevant and important information of specifications and implementations are extracted through invoking their respective probing operations which perform queries on the state variables. The probing operations at the abstract level as well as at the concrete level have nothing to do with the functionalities of the system, and they only extract out important state aspects by querying the system state. The probing operations are actually instrumentations to both specifications and implementations.

The formal specification-based monitoring technique proposed in this thesis gets required information about dynamic behaviors of the formal specification and concrete implementation of the target system through animating and debugging respectively, rather than by embedding any instrumentation code into the target system or by annotating the concrete implementation with extra formal specifications. Consequently, our formal specification-based runtime monitoring technique will not alter the running environment and the dynamic behaviours of the target system which is being monitored. Moreover, our monitoring technique realizes the

clear separation between the implementation-dependent description of monitored object and the highly abstract formal specification of it, which allows the reuse of the formal requirement specification when changes happen to the implementation of the target system.

2.3 Regression Testing

Regression testing is the process of validating modified software to provide confidence that the changed parts of the software behave as intended and that the unchanged parts of the software have not been adversely affected by the modifications [37]. It is an important and expensive software maintenance activity.

Typically, the regression test proceeds as follows [35], where P is a program, P′ is a modified version of P, and T is a test suite for P.

1. Select T′ ⊆ T, a set of test cases to execute on P′.
2. Test P′ with T′, establishing the correctness of P′ with respect to T′.
3. If necessary, create T″, a set of new functional or structural test cases for P′.
4. Test P′ with T″, establishing the correctness of P′ with respect to T″.
5. Create T‴, a new test suite and test execution profile for P′, from T, T′, and T″.

The process of regression testing involves a few problems: namely, regression test selection (step 1), test suite augmentation (step 3), test suite execution (steps 2 and 4) and test suite maintenance (step 5).

One characteristic that distinguishes regression testing from development testing is the availability of existing test suite that was used to test the original version of the system. Reusing the existing test suite can reduce the cost and effort required by regression testing. However, rerunning all of the test cases in the existing test suite may take pretty high cost because the original test suite could be large, and the time and effort required to rerun all the test cases may be excessive. Moreover, some of the test cases in the existing test suite may be obsolete to the modified version of the system and cannot be used for regression testing. Consequently, during the process of regression testing, the foremost problem which needs to be addressed is the *regression test selection problem* [37], i.e., to restrict testing efforts to a subset of the existing test suite. A solution to this problem is to apply regression test selection techniques to select a proper subset of the test suite for regression testing. A safe regression test selection technique is one that, under certain assumption, selects every test case from the original test suite that can expose faults in the modified program [83].

To date, several safe regression test selection techniques have been developed for retesting software developed with procedural programming languages or object-oriented programming languages [9, 19, 59, 84, 106, 110, 48, 85, 109, 37]. Rothermel *et al.* [85] presented a regression test selection technique for C++ software. The

technique constructs control flow representations for classes and programs that use classes and handles both structural and nonstructural modifications and processes multiple modifications with a single application of the algorithm. Rothermel and Harrold also had presented a regression test selection for C++ software based on walks of program dependence graphs in [82]. This technique is more efficient than which is presented in [82]. Harrod *et al.* [37] presented the first safe regression test selection technique that handles the features of Java language. Compared to Rothermel, Harrold and Dedhia's technique for C++ [85] and White and Abdullah's firewall technique [109], the technique presented in [37] is more precise, can be applied to incomplete programs, handles exception-handling constructs, and provides a new method for handling polymorphism. Wong *et al.* [111] proposed a technique that combines modification, minimization and prioritization-based selection using a list of source code changes and the execution traces from test cases run on previous versions. This technique seeks to identify a representative subset of all test cases that may result in different output behavior on the new software version.

These existing techniques are code based. They select test cases for the regression testing, from the original test suites, using the information gathered by code analysis. The main difference between our formal specification-based approach and those existing techniques is that our approach is strictly specification-based and independent of the programming language that is used to implement the specification. Therefore, it is capable of selecting test cases for the detection of faults caused

by the conditions where the specifications for the software have been changed, but the code modifications necessary to implement the changed specifications have not been made.

2.4 AOSD and Formal Methods

Aspect-oriented software development(AOSD) is a promising technology that supports multi-dimensional separation of concerns throughout the software development cycle [33, 50, 100].

As software systems become increasingly large, complex and distributed, traditional development techniques cannot effectively modularize the global concerns of the systems, such as synchronization, distribution, security, coordination and persistence. These concerns normally cut across several parts of the systems, and often overlap. AOSD addresses the crosscutting concerns by providing means for systematic identification, separation, representation and composition. Crosscutting concerns are encapsulated in separate modules, known as aspects, so that localization can be promoted. The main benefit of AOSD is that it improves system modularization, by reducing scattered and tangled code, avoiding the typical mixing between functional and extra-functional properties, enabling a better code evolution management. This results in the remarkable reduction of development, maintenance and evolution costs.

A few methodologies for aspect-oriented requirement analysis, architecture de-

sign, and graphical visualizations have been proposed [10, 86, 94, 113]. Theme approach [10] provides support for aspect-oriented development at requirement level and design level. Theme/Doc provides views of requirements specification text, exposing the relationship between behaviors in a system at requirement level. Theme/UML allows a developer to model features and aspects of a system, and specify how they should be combined at the design level. Interaction diagram-based Join Point Designation Diagrams (JPDDs) [95] have been proposed as a modeling approach especially dedicated to the graphical representation of join point selections. In particular, the notation provides graphical means to visualize joint point queries based on the lexical properties of program elements as well as based on the dynamic and structural context they occur in. However, comprehensive technique supports for modeling and design are still far from sufficient, and more improvements are in demand.

Among the efforts aiming to provide a suitable design notation for the design of aspect-oriented programs, most of them are proposals to extend UML to present graphical notations. The first proposal to extend the UML with concepts for the design of aspect-oriented programs comes from Suzuki and Yamamoto [98]. In their approach, a new UML meta-class named "aspect" is introduced, which is related to base classes using a UML realization relationship. Clarke *et al.* [20, 21] extended the UML with a new design concept - composition patterns. Composition patterns are UML templates for UML packages which are bound to actual classes and operations by means of a special binding compositional relationship. Stein

et al. [94] presented aspect-oriented design model (AODM), which extends the UML with the aspect-oriented design concepts as they are specified in AspectJ. It provides suitable representations for all components of an aspect as well as for the aspect by extending existing UML concepts using UML's standard extension mechanisms. It also implements AspectJ's weaving mechanism in the UML and specifies a new relationship signifying the crosscutting effects of aspects on their base classes.

Some researchers also have tried to extend mathematics and/or logic based formal specification notations to support aspect-oriented program design and verification. Ubayashi and Nakajima [104] employed the feature-oriented modeling method and the VDM-based formal design with the notion of the aspect and proposed AspectVDM. AspectVDM is aspect-oriented extension to VDM, following the idea of Join Point Model in AspectJ. Zhao and Rinard [116] proposed Pipa, which is a behavioral interface specification language tailored to AspectJ. As a simple and practical extension to the Java Modeling Language(JML), Pipa uses the same basic approach as JML to specify AspectJ classes and interfaces, and extends JML to specify AspectJ aspects with a few new constructs. The work most close to ours is what have been proposed by Yu *et al.*. They proposed AspectZ [115], an aspect-oriented extension to Z. In a similar way, Yu *et al.* [114] introduced the concept of join point, pointcut, advice and aspect to Object-Z. AspecTCOZ, the formal specification notation proposed by us, is different from the work by Yu *et al.* in two main ways. Firstly, in AspecTCOZ, advice is defined with the assistance of

operation schema, which provides the system designers with the ability to abstract and encapsulate the description of what to do at the join point, and the ability to reuse this formal description. Secondly, with the strength of TCSP in modeling process control and real-time interactions, which is preserved in TCOZ, the temporal order between pointcut and advice can be described clearly and concisely in AspecTCOZ notation.

Chapter 3

Formal Specification-based Software Monitoring

In this chapter, a formal specification-based software monitoring technique is presented.

3.1 Introduction

Software monitoring technique analyzes and determines whether the observed software behavior complies with specified requirements. With the capabilities of detecting, diagnosing and recovering from software faults, it provides additional defense against catastrophic software failure.

Recently, there has been increasing attention from the research community to the development of techniques and tools for runtime monitoring of traditional softwares [11, 40, 56]. In order to monitor software system, some of the existing monitoring techniques add instrumentation codes to the target program to collect required data about dynamic behaviors of the target program while it is running. Others achieve monitoring by annotating the concrete implementation with extra formal specifications to obtain required dynamic information about the target program.

There are a few disadvantages in adding instrumentation codes to target program and annotating target program with extra formal specifications. Adding instrumentation code is itself a difficult task involving all the complexities of programming. Moreover, it generally leads to changes in the program; it raises the possibility that through collecting information to analyze target system behavior, the monitoring system is actually altering that behavior of the target system. Annotating the concrete implementation with extra formal specifications leads to the lack of separation between the concrete implementation of target systems and the high-level

requirements specification of them.

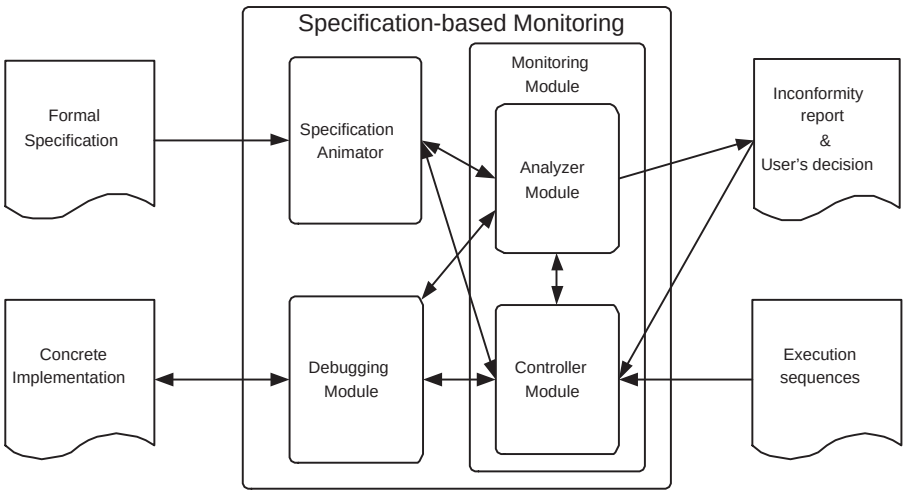

Figure 3.1: Formal specification-based software monitoring system

In this chapter, we propose a novel formal specification-based software monitoring technique. The key idea of our approach is to build a linking system (monitoring module) which connects a specification animator and a program debugger. As shown in Figure 3.1, in our specification-based monitoring approach, a specification animator is used to exhibit the dynamic behavioural properties of the formal specification; a debugger is used to extract the information about the dynamic behavior of the concrete implementation. The monitoring module controls the specification animator and the debugging module so that the concrete implementation will run in parallel with the animation of the formal specification. Meanwhile, based on the information obtained from the specification animator and the debugging module, the monitoring module will dynamically check the conformance of the concrete implementation with the formal specification.

The proposed formal specification-based software monitoring technique does not embed any instrumentation codes to the target system, neither does it annotate the target system with any formal specifications. It can detect errors in a timely manner, prevent the errors from propagating, and help the developers and users of the system to take recovery actions before critical failure happens. With the ability of continuously monitoring and checking a running system with respect to its formal specification, the software monitoring technique is a good candidate to be used as a complementary technique to traditional formal verification and software testing.

The remainder of this chapter is organized as follows. Section 3.2 introduces specification animation. Section 3.3 presents an overview of the formal specification-based software monitoring technique; describes a prototype that we have developed for demonstrating the software monitoring technique; and discusses a few technical challenges that we have encountered in the development of the prototype. Section 3.4 analyzes the merits and limitations of the formal specification-based monitoring technique and discusses the impacts of it on traditional software testing. Finally, section 3.5 concludes this chapter.

3.2 Specification Animation

Specification animation exhibits the dynamic behaviourial properties of formal specification. It not only gives the specification designers a way to test whether

their specifications behave as expected, but also validates the behavior of formal specifications with the end users. The specification animation technique has been used to assist systematic validation of formal specifications [75, 76].

In the last decade, several animation tools have been developed for executing and interpreting formal specifications automatically. For example, PiZA [43] is an animator for Z, and Possum [41, 42] is an animator for Z and Z-like specification language.

The animation tool used in our prototype monitoring system is Jaza [105]. It is an animator for Z, which has a strong support for quantifiers and various less-often-used Z constructors(such as μ, λ, θ terms). It provides more efficient and convenient evaluation of schemas on ground data values; and it has the ability to search for example solutions of a schema or predicate. Jaza supports at least twelve different representations of set. And this makes it more advanced in its execution than other animators for Z. Moreover, Jaza can handle not only unpredictable performance characteristics but also nondeterministic schemas.

3.3 Formal Specification-based Software Monitoring

This section presents the formal specification-based software monitoring technique that we have proposed. Section 3.3.1 presents an overview of the proposed monitoring technique. Section 3.3.2 describes a prototype monitoring system that we have

developed for demonstrating the technique. Section 3.3.3 discusses a few technical challenges that we have encountered in the development of the prototype and the ways that we have surmounted them.

3.3.1 Overview of the Technique

Given the concrete implementation and the formal specification of a system, our formal specification-based monitoring technique will dynamically check the conformance of the concrete implementation with the formal specification.

The overview picture of the monitoring technique has been shown in Figure 3.1. With the specification animator, our specification-based monitoring technique gets the information about the dynamic behavioural properties of the formal specification through specification animation; and with the debugging module, the information about the dynamic behavior of the concrete implementation is gathered through program debugging. Taking the execution sequences provided by the user as input, the monitoring module controls the specification animator and debugging module so that the concrete implementation will run in parallel with the animation of the formal specification. Meanwhile, based on the information obtained from the specification animator and debugging module, the monitoring module provides judgement on the conformance of the concrete implementation with the formal specification. If any inconformity is found, it will be reported to the user. With the inconformity report, the user needs to make a decision about how to deal with such an inconformity. Then, the monitoring system will continue its work according

to user's decision.

In our monitoring technique, the monitoring module functions as an external observer of the target system. Moreover, the monitoring module is designed to monitor the target system and respond in a timely manner while the target system is running. This means that our monitoring technique not only gathers information, but also dynamically interprets the gathered information and responds appropriately.

3.3.2 A Prototype

To demonstrate our formal specification-based monitoring technique, we have implemented a prototype monitoring system. The prototype monitoring system works with the formal specification written in the Z formal language and the concrete implementation programmed in Java programming language. In our monitoring system, the animator used for animating the formal specification is Jaza [105], the debugger used for extracting required information from the execution of Java program is ***jdb*** [3]. ***jdb*** is the debugger supplied by Sun in the Java Developer's Kit (JDK); and it is implemented using the Java Debugger API.

The monitoring system can work in two different modes: debugging mode and running mode. After the formal specification and concrete implementation have been loaded to the monitoring system, the system will extract the operations and state variables defined in the formal specification, and the methods and class variables

defined in concrete implementation. With the assistance from users, the monitoring system will match the operations defined in the formal specification with corresponding methods defined in the concrete implementation; and it will also perform similar matching for the state variables and corresponding class variables. Then, the user needs to decide whether the monitoring system will work in the debugging mode or the running mode.

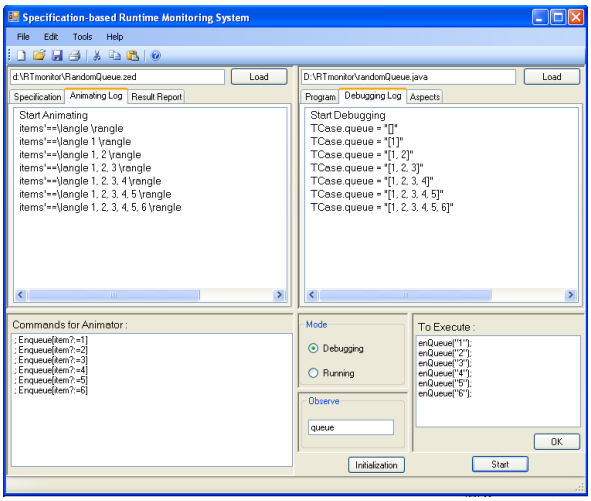

Figure 3.2: Debugging mode

In the debugging mode, as shown in Figure 3.2, after matching the operations/state variables in specification with methods/class variables in implementation, the user inputs all of the methods which are expected to be executed into the monitoring system as a whole sequence; and the system will automatically generate the sequence of corresponding running commands for the animator. Then, the system starts the dynamic checking of the conformance between behavior of the concrete implementation and the behavior of the formal specification animation . In the debugging mode, our specification-based monitoring system can serve as an effective

dynamic test execution and test result checking tool.

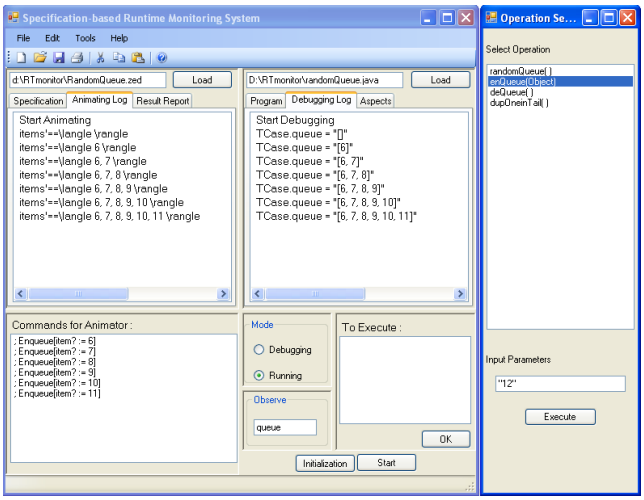

Figure 3.3: Running mode

In the running mode, as shown in Figure 3.3, after matching the operations/state variables in the specification with the methods/class variables in the implementation, the user manipulates the monitoring system by indicating the methods to be executed and inputting parameters (if necessary) in a one-by-one way rather than inputs all of the methods which are expected to be executed into the monitoring system as a whole sequence, as in the debugging mode. The commands for executing corresponding operations will be automatically generated for the animator. The monitoring system will then check the running result of the implementation execution with the corresponding specification animation result to figure out whether there is an inconformity.

In the running mode, the user indicates what will be executed next in a step-by-step way. Thus, the system can achieve the on-the-flying monitoring of concrete imple-

mentations against formal specifications. In the running mode, the user guided execution sequence selection can be easily adapted to connect to a runtime execution system to achieve the monitoring of reactive safety-critical systems.

3.3.3 Technical Challenges

Matching of structural elements between concrete implementation and formal specification

There exist structural and semantic gaps between formal specification and programming languages. On the one hand, the formal specification of a software/hardware system usually describes the system at a high level of abstraction with a formal language. Generally, the formal specification is very expressive and includes many rich abstract data types. Those abstract date types have no data representation specified, and the implementations of their operations are also kept abstract [87]. On the other hand, to implement the specification, those abstract data types must be implemented by the existing data types in the programming language. There may be various potential concrete representations in the implementation for a certain abstract data type in the specification. We take the *queue* that we have introduced in Chapter 2 as an example. In the specification of the *queue* which is redisplayed in Figure 3.4, the type of state variable *items* is *sequence*. Suppose the programming language we use to implement the *queue* is Java, we may use the `java.util.Vector` classes to implement the abstract data type *sequence*. And, alternatively, we can also use `ArrayList` to achieve the implementation successfully.

$size == 10$

Queue

$items : \operatorname{seq} \mathbb{N}$

$\#items \leq size$

InitQueue

$Queue'$

$items' = \langle\rangle$

Enqueue

$\Delta Queue$
$item? : \mathbb{N}$

$\#items < size$
$items' = items \frown \langle item? \rangle$

Dequeue

$\Delta Queue$
$item! : \mathbb{N}$

$\#items > 0$
$item! = head\ items$
$items' = tail\ items$

DupOneinTail

$\Delta Queue$
$item! : \mathbb{N}$

$\#items > 1 \land \#items < size$
$item! \in \operatorname{ran} items \land item! \neq head\ items$
$items' = items \frown \langle item! \rangle$

Figure 3.4: *Queue* in Z notation

Therefore, to check whether the concrete implementation is conformable with the formal specification, the first thing that needs to be done is to match the variables, data types and operations in formal specifications with their corresponding counterparts in concrete implementations. This matching must take into account differences in the level of abstraction in the formal specification language and programming language, because formal specifications do not address many of the details addressed by the implementation. As shown in Figure 3.5, our monitoring

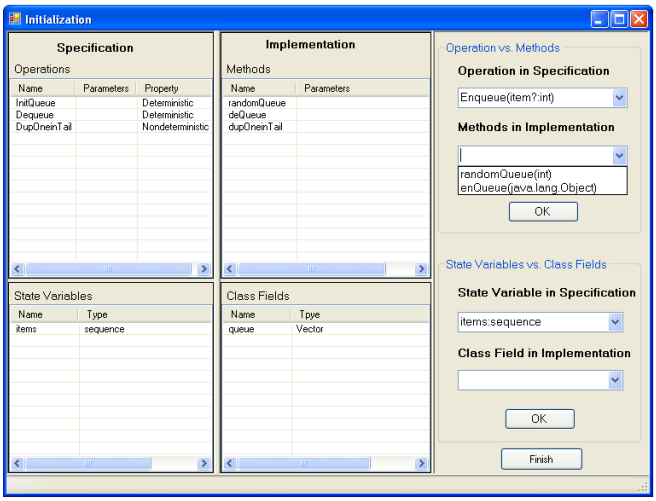

Figure 3.5: Matching between concrete implementation and formal specification

system automatically extracts the operations and state variables defined in the formal specifications, and the methods and class variables defined in concrete implementations. For any operation or state variable appointed by the user, the system lists all the possible matching candidates for it. For operations, the matching candidate methods are selected in terms of signatures (i.e., number and type of parameters) of methods and operations. For state variables, the matching candidate class variables are selected in terms of the types of the state variables and class variables. In this way, the system reduces remarkably the effort required for the user to find the right matches. Thus, the users can finish the matching based on their knowledge of both formal specification and the concrete implementation more efficiently.

Inconformity of concrete implementations to formal specifications

If the concrete implementation is not conformable with the formal specification, the monitoring system will detect and report such an inconformity. What should the monitoring system do after an inconformity has been revealed? If we let the animator keep running, all of the subsequent judgement will apparently be *inaccurate* due to the carry-over of inconformity from the previous execution. Therefore, we propose two choices to the user. The first choice is to stop the monitoring process whenever an inconformity is detected, and let the user to fix the problem in the implementation. However, this choice will reduce the effectiveness of the system as only one inconformity can be found at one execution round. The second choice is to reinitialize the animator with the corresponding execution result from the implementation to keep the animation and the implementation in the same state before the next operation/method is executed. This choice is based on the assumption that the execute result of the implementation is correct.

For example, the Java code shown in Figure 3.6 is supposed to implement the *queue* which is specified by the Z specification displayed in Figure 3.4. After loading the specification and implementation to the monitoring system and finishing necessary matching between the specification and the implementation, we start the monitoring process. As shown in Figure 3.7 , after the execution of the operation of deleting an item from the queue, the monitoring reports an inconformity and points out that the operation *Dequeue* is not implemented correctly. When we checked the Java code, we found that the method deQueue actually deletes the last item from

```
class Queue {
    private Vector queue;
    Queue (){queue = new Vector();}
    public void enQueue (Object item) {
        queue.addElement(item); }
    public Object deQueue () {
        Object obj = null; int last;
        if (! queue.isEmpty()) {
                  obj = queue.lastElement();
                  last = queue.size();
                  queue.removeElementAt(last-1);
        } return obj;
    }
    public Object dupOneinTail() {
        int n, i; Object obj = null;
        n = queue.size();
        if(n>1){
             i = (int)(n*Math.random( ));
             obj = queue.elementAt(i);
             queue.addElement(obj);
        } return obj;
    }
   }
```

Figure 3.6: Class *Queue* in Java

the queue, while the specification demands that the operation *Dequeue* delete the first item from the queue. When the monitoring system reports an inconformity, it provides the two choices:(1) stop monitoring, (2) reinitialize the animator with the corresponding execution result from the implementation, as shown in Figure 3.7. If we choose to reinitialize the animator, the state variable *items* will be set to $\langle 6, 7, 8, 9, 10 \rangle$ before the next operation is executed and the monitoring system can continue working. Alternatively, we can choose to stop monitoring, and fix the problem in the implementation.

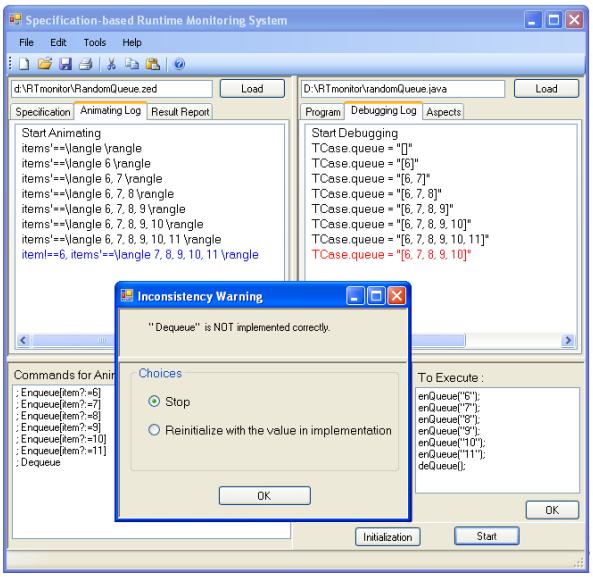

Figure 3.7: Inconformity occurs

Nondeterministic operations in the formal specification

An issue that complicates the matter is specification nondeterminism, i.e., the specification may involve the definition of nondeterministic operations, the execution of which may lead to more than one possible results. When encountering a nondeterministic operation in the specification, the animator will present a legal but stochastic result.

However, the implementation is always deterministic. It may present a result that is legal to the specification but different from that of the animator. If the monitor only performs simple comparison of the two results, it will definitely make a wrong judgement. Therefore, how to make the monitoring system judge correctly in a nondeterministic situation is one of the major challenges of the formal specification-based monitoring technique. A possible solution for handling such nondeterminism

is to make the animator present all the possible legal results and let the monitoring system take all of them into consideration when comparing the results between the implementation and the specification to avoid misjudgement. However, this approach may be time consuming and may considerably increase space complexity when the number of possible results is large.

Our specification-based monitoring system provides a mechanism for the user to indicate whether the operation is deterministic or nondeterministic when matching an operation in the specification with a method in the implementation. When a nondeterministic operation is encountered in the process of monitoring, the specification animator will be demanded to present one possible result at a time. The monitoring module compares the result from specification animation with the corresponding result from implementation execution. If they are conformable, the monitoring system will decide that the implementation of this operation is correct. If not, the specification animator will continue to present another different possible result, the monitoring system will perform another comparison, and the monitoring system will repeat the above process till the results from both sides are conformable or all of the possible animation results are presented. In the latter case, where the result from implementation is not conformable with any of the possible results from the animator, the system will make the judgement that the operation is not implemented correctly with respect to the formal specification.

In the Z specification of a *queue* shown in Figure 3.4, there is a nondeterministic operation – *DupOneinTail*. Based on the semantics of the Z formal language, we

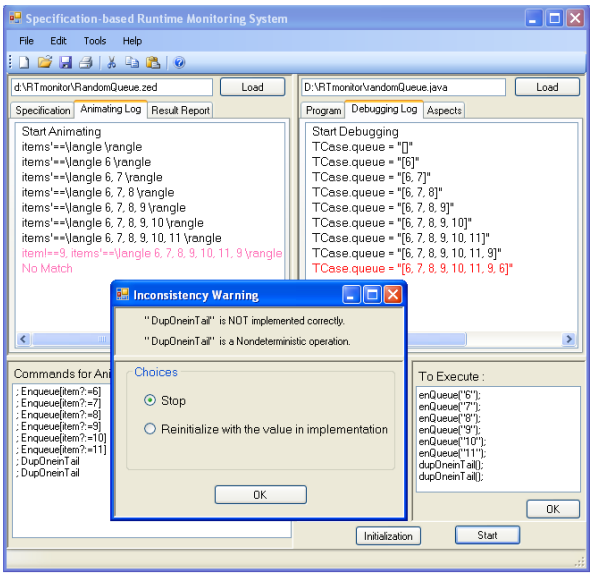

Figure 3.8: Handle nondeterminism.

can figure out that there would be more than one possible results for a single execution of the operation *DupOneinTail*. In Figure 3.8, it can be seen that method dupOneinTail, which is supposed to implement the operation *DupOneinTail*, appears twice in the execution sequence. When it is executed for the first time, it selects "9", which is the fourth element of the queue, and attaches it to the tail of the queue. This is feasible according to the formal specification and the specification animator certainly finds a corresponding result that matches it. The monitoring system will not report any inconformity. It only changes the color of the result in the window to indicate that the operation *DupOneinTail* is nondeterministic. However, when the method dupOneinTail is executed for the second time, it selects "6", which is the head of the queue, and attaches it to the tail of the queue. This is infeasible according to the formal specification. The specification animator exhausts all the possible legal results and could not find a match. Therefore, the

monitoring system concludes an inconformity at this time, and reports it to the user.

By examining the original implementation displayed in Figure 3.6, we find that the inconformity is indeed caused by the dupOneinTail method which does not implement one of the preconditions (*i.e.*, the selected item cannot be the head of the queue) in the specification.

3.4 Discussion

The proposed formal specification-based software monitoring technique aims at providing a feasible and affordable approach for the improvement of software reliability and quality. It provides assurance that the target system is running correctly with respect to its formal specification by checking the correctness of the execution of the target program at runtime. Now, we analyze the merits and limitations of it and discuss the impacts of it on software testing.

3.4.1 Merits

The formal specification-based monitoring technique proposed in this chapter gets required information about dynamic behaviors of the formal specification and concrete implementation of the target system through specification animating and program debugging respectively. It does not embed any instrumentation code into the target system, therefore, it will not alter the running environment and the

dynamic behaviors of the target system which is being monitored. Meanwhile, it does not annotate the concrete implementation with any extra formal specifications neither. Consequently, it allows the reuse of a highly abstract formal requirement specification when changes happen to the implementation the target system.

Furthermore, the formal specification-based monitoring technique always monitors the current state of the system, continuously checks the conformance of implementation with formal specification and reports any detected inconformity immediately whenever any failures happen. Therefore, by ensuring that the current execution is conformable with its requirements at runtime, formal specification-based monitoring can provide the developers and users with much higher confidence in the software than traditional testing. Besides, although the formal specification-based monitoring technique is weaker than formal verification in terms of the ability to guarantee software correctness, it provides a dynamic verification technique by checking that the actual execution of a system is conformable with the expectation described by the formal specifications. Rather than checking that the design model of the system satisfies some properties, as formal verification does, the formal specification-based monitoring checks that the results of particular computations when the system is executed are correct with respect to the formal specification. Thus, formal specification-based monitoring escapes from the state-space explosion problem that limits the scalability of formal verification techniques. Therefore, the formal specification-based monitoring technique can serve as a complement to traditional testing and formal verification techniques in software quality assurance.

3.4.2 Limitations

The limitation of the proposed formal specification-based monitoring approach is often related to the capabilities of the specification animator. It is very likely that some aspects of the formal specification can not be simulated by the animator. For example, Jaza, the animator that we used, does not support the type of *bag* and generic constructs in Z notation. It can not handle user-defined infix/prefix/postfix functions or relations neither.

There are also limitations resulting from the manner in which specifications are expressed. State-based specification languages such as Z and VDM do not allow the developer to easily express temporal and concurrent properties of a system, while process-oriented specification languages such as CSP and CCS are generally poor at expressing the structure and state of a system. So far, we focus on working with state-based specification languages. In the future, the formal specification-based monitoring system that we have developed will be extended to verify temporal and concurrent aspects of software systems.

Furthermore, the description granularity of the formal specification languages determines the granularity of inconformity detection that the proposed formal specification-based approach can achieve. For example, the Z specification language specifies a system by describing its state and the way in which the states can be changed. The formal specification of a system in Z notation consists of a sequence of state schema and operation schemas. Therefore, the monitoring system which works with specifications in Z notation can only detect which operation schema in the specification

is not implemented as expected, but can not determine which predicate in the specification is violated. Moreover, there are syntactic and semantic gaps between specification and programming languages. Usually, implementations address much more details than formal specifications do. Consequently, the proposed monitoring approach can detect which method in the program is not implemented correctly with respect to the corresponding operation schema in the formal specification, but can not precisely locate the particular statement in the implementation that contains the error.

Additionally, some kinds of properties of a system can not be verified at runtime with the proposed monitoring approach. For example, safety properties which state that *something bad will never happen* can not be verified for infinite state systems because the monitoring system can only check finite state traces.

3.4.3 Impacts on Software Testing

Software monitoring has some appealingly positive impacts on traditional software testing. Software testing activities typically include four main steps: generating testing inputs, creating expected outputs, running software with test inputs, and verifying actual outputs. To reduce the burden of manually creating test inputs, some test input generation tools [4, 2, 24] are used to generate test inputs automatically. However, the expected outputs for these test inputs are still missing, and it is almost infeasible for developers to create expected outputs for the large number of generated test inputs. Even if developers are willing to invest efforts in

generating expected outputs, it is expensive to maintain these expected outputs since some of the expected outputs need to be updated whenever the program is changed [51, 73].

Without the expected outputs available, it is often expensive and prone to error for developers to manually verify the actual outputs and it is limited in exploiting the automatically generated test inputs by only checking whether the program crashes or throws uncaught exceptions. Although, in regression testing, the actual outputs of a new version can be compared with the actual outputs of its previous version, behavioral differences between the two versions might not be propagated to the observable outputs that are compared between versions.

On all accounts, traditional software testing is hampered by the need to independently compute and verify the expected outputs for each test case.

By assisting traditional software testing with runtime monitoring, there is no need to compute expected outputs before test executions or verify the actual outputs after test executions because, with specific inputs, the monitor system will make a judgement on the correctness of the actual outputs based on the results from the animation of formal specifications. Furthermore, the reinitialization mechanism, which is designed to handle the situation where an inconformity is detected, enables continuous testing in which the system can keep running until the test input sequence ends rather than stops when an error is detected. Consequently, the formal specification-based runtime monitoring technique has the potential of greatly reducing the cost of testing and significantly increasing the effectiveness

of testing. The reason is that, with the assistance of runtime monitoring system, testing is no longer limited to small sets of test inputs, to which expected outputs are available, and can detect more than one error with a test input sequence and one round execution of the system which is under test.

3.5 Conclusion

This chapter presents a formal specification-based software monitoring technique. With the formal specification and concrete implementation of the target system, our specification-based monitoring technique uses a specification animator to exhibit the dynamic behavior of the formal specification, uses a program debugger to extract required information about the dynamic behavior of the concrete implementation, and checks the conformance of the concrete implementation with the formal specification, based on the information from the animator and the debugger.

Our monitoring technique gets required information about dynamic behaviors of the formal specification and concrete implementation of the target system through animating and debugging respectively, rather than by embedding any instrumentation code into the target system or by annotating the concrete implementation with extra formal specifications. Consequently, our formal specification-based runtime monitoring technique will not alter the running environment and the dynamic behaviors of the target system which is being monitored. Moreover, our monitoring technique realizes the clear separation between the implementation-dependent

description of monitored object and the highly abstract formal specification of it, which allows the reuse of the formal requirement specification when changes happen to the target program.

Moreover, as an runtime monitoring technique, our formal specification-based monitoring technique dynamically gathers required information, interprets the gathered information and responds appropriately in a timely manner as the target system runs. Therefore, it is competent for monitoring reactive safety-critical systems.

To sum up, our formal specification-based monitoring technique can contribute to the dependability, correctness and robustness of the target system; and it also can contribute to effective dynamic test execution and test result checking.

Chapter 4

Monitoring System Case Studies

In this chapter, the application of the formal specification-based software monitoring technique is demonstrated with case studies.

4.1 Introduction

In this chapter, we first demonstrate the application of our specification-based monitoring technique with the railway track line automatic blocking scheme [14] of a railway control system. Due to the inherent characteristics of a reactive safety-critical system, it should be monitored by an independent monitoring system which can make quick and precise determination whether the observed behaviours are acceptable or not. We will illustrate that our monitoring system satisfies the above requirement, and can be applied to ensure the continuous correct behaviours of a reactive safety-critical system.

Furthermore, we also demonstrate the application of our specification-based monitoring technique with the verification of a robotic assembly system which has been studied in [6, 7, 28] and extended to be used for the assembly of spacecrafts in outer space during NASA's ANTS mission [25, 26, 44].

4.2 A Railway Control System

In the railway system, in order to avoid that the trains which run in the same direction on the same track crash into one another "*from behind*", the track is divided into segments with visible signals at the segment connections. The trains may pass a signal if there are no trains in the approaching segment (signal is set to green), or if it is some while ago that a previous train passed the segment (signal is then set to yellow). Otherwise, if the approaching segment is occupied by another

train, the current train is blocked as the signal is set to red.

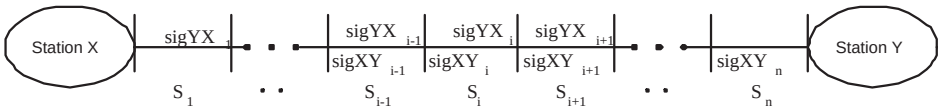

Figure 4.1: Track line signalling

As shown in Figure 4.1, line l which connects exactly two stations: station X and station Y, is usually divided into segments $l = \langle s_1, s_2, ..., s_{i-1}, s_i, s_{i+1}, ..., s_n \rangle$. A line l can be in one of three possible states: OpenXY, OpenYX and Close. Each segment can be in two states: Free or Occupied. Segment s_i is in the state of Free when no train is detected in the segment. Otherwise, segment s_i is in the state of Occupied.

For each inner segment s_i, where i $= \langle 2, ..., n-1 \rangle$, there are two signals $sigXY_i$ and $sigYX_i$ which are for the two opposite directions of travel. Each signal is associated with four possible states: Red, Yellow, Green and Off.

Signal $sigXY_i$ is in Red state when line l is in OpenXY state and segment s_i is in Occupied state. It is in the Green state when line l is in OpenXY state and both segment s_i and s_{i+1} are in Free state. It is in Yellow state when line l is in the OpenXY state, segment s_i is in Free state and segment s_{i+1} is in Occupied state. It is in the Off state, when line l is in OpenYX or Closed state. Correspondingly, it is easy figure out the situations when signal $sigYX_i$ will be in the four different states.

For the first segment s_1 and the last segment s_n, there is only one signal $sigYX_1$

and signal $sigXY_n$, respectively. The signals in the opposite directions ($sigXY_1$ and $sigYX_n$) are controlled manually, or by interlocking in the station [14].

The main body of the formal specification that describes the railway track line automatic blocking scheme in Z notation is shown as follows. A complete version of the formal specification is displayed in Appendix A.

trnOneEnter

$\Delta Track$

$trnOne.pos = OffTrack \land status = OpenAB \land trnOne.dir = MoveAB$
$segA.sigAB = Green \lor segA.sigAB = Yellow$
$segA'.status = Occupied \land segA'.sigAB = Red$
$trnOne'.pos = A \land trnOne'.dir = trnOne.dir$
$trnTwo' = trnTwo \land status' = status \land segB' = segB$

trnTwoEnter

$\Delta Track$

$trnTwo.pos = OffTrack \land status = OpenAB \land trnTwo.dir = MoveAB$
$segA.sigAB = Green \lor segA.sigAB = Yellow$
$segA'.status = Occupied \land segA'.sigAB = Red$
$trnTwo'.pos = A \land trnTwo'.dir = trnTwo.dir$
$trnOne' = trnOne \land status' = status \land segB' = segB$

trnOneMovetoB

$\Delta Track$

$trnOne.pos = A \land status = OpenAB \land trnOne.dir = MoveAB$
$segB.sigAB = Green \land segA'.status = Free$
$segA'.sigAB = Yellow \land segB'.sigAB = Red \land segB'.status = Occupied$
$trnOne'.pos = B \land trnOne'.dir = trnOne.dir$
$trnTwo' = trnTwo \land status' = status$

trnTwoMovetoB

$\Delta Track$

$trnTwo.pos = A \land status = OpenAB \land trnTwo.dir = MoveAB$
$segB.sigAB = Green \land segA'.status = Free$
$segA'.sigAB = Yellow \land segB'.sigAB = Red \land segB'.status = Occupied$
$trnTwo'.pos = B \land trnTwo'.dir = trnOne.dir$
$trnOne' = trnOne \land status' = status$

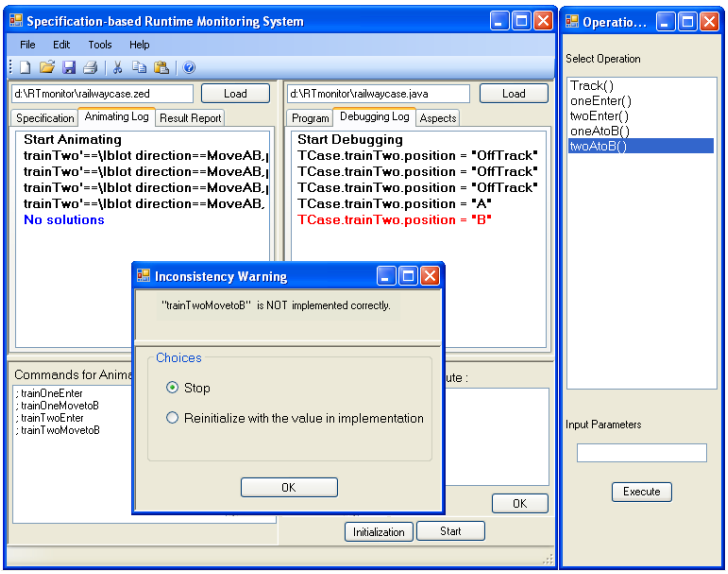

Figure 4.2: Monitor a railway control system

The Java programming language displayed in Appendix B is supposed to implement the railway track line automatic blocking scheme. To check the conformance of the implementation with the formal specification, we load the formal specification and concrete implementation of the railway track line automatic blocking scheme to our specification-based monitoring system. As shown in Figure 4.2, the monitoring system works in running mode where the user will indicate what will be executed next in a step-by-step way. Figure 4.2 also shows that the monitoring system finds an inconformity and reports that operation *trainTwoMovetoB* is not implemented correctly. By checking the operation sequence, we know that the second train enters a segment while the first train is already on it. This is not allowed by the railway line automatic blocking scheme. The part of the formal specification which describes the above property of the railway system is shown as follows.

trainTwoMovetoB

$\Delta Track$

$trainTwo.position = A \land status = OpenAB$
$trainTwo.direction = MoveAB$
$segmentB.signalAB = Green \lor segmentB.signalAB = Yellow$
$segmentA'.status = Free \land segmentA'.signalAB = Yellow$
$segmentB'.signalAB = Red \land segmentB'.status = Occupied$
$trainTwo'.position = B \land trainTwo'.direction = trainOne.direction$
$trainOne' = trainOne \land status' = status$

By checking the concrete implementation in Java, it is found out that the cause of the inconformity is in method twoAtoB. As shown by the following code, the method twoAtoB which is supposed to implement the operation *trainTwoMovetoB* defined in the formal specification, does not correctly implement the checking of signal as described by the specification. In the *if* statement of method twoAtoB, there should have been a substatement which checks the variable corresponding to the signal at the segment connection.

```
public void twoAtoB(){
   if(status == TrackState.OpenAB
         && trainTwo.position == TrainPosition.A
             && trainTwo.direction == TrainDirection.MoveAB)
   {
     segmentA.status = SegState.Free;
     segmentA.signalAB = SigState.Yellow;
     segmentB.signalAB = SigState.Red;
     segmentB.status = SegState.Occupied;
     trainTwo.position = TrainPosition.B;
    }
   }
```

4.3 A Robotic Assembly System

A robotic assembly system has been studied in [6, 7, 28]. As shown in Figure 4.3, the assembly unit consists of a robot system, a conveyor belt, and an assembly-tray. The conveyor belt normally carries the objects to be assembled. The objects, n of each kind, are placed on the conveyor belt and the robot may pick up an item from the conveyor belt at a prespecified location. It is expected that for any arbitrary placement of the objects, n of each kinds, on the conveyor belt the robot would assemble the objects in order and produce n assemblies. The robot system consists of two arms, a vision system and a stack. The vision system can recognize the objects to be assembled and record the number of the objects of each kind that have been recognized. The stack is for temporarily storing the objects. The vision system is continuously focused on the conveyor belt so that the scanning and recognition begins when the objects on the conveyor belt enters the camera's view and the belt is stopped. When the vision system recognizes an object, the left arm or the right arm is activated so as to pick that item from the belt. Initially, the arms are free and the stack is empty. Whenever both arms are free and the stack is empty, and the vision system recognize an object then the left/right arm picks up the item from the conveyor belt and begins the process of assembly. If the object on the left/right arm is the same as a part of the half-assembled product, the object on the left/right arm will be pushed into the stack; otherwise, the object will be assembled. If the left/right arm is free but the stack is not empty and the top item of the stack is not same as any part of half-assembled product, the left/right arm

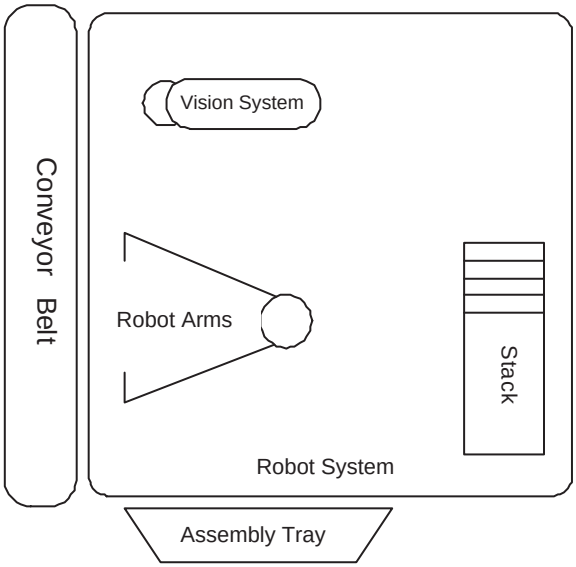

Figure 4.3: Robotic assembly system

picks up (pops) an object from the stack. If the assembly of one piece of product has been completed, the product will be released and placed on an assembly-tray.

Autonomous Nano-Technology Swarm (ANTS) mission [25, 26, 44] is one of NASA's future space exploration missions which use intelligent swarms of spacecrafts [102, 103]. During the ANTS mission, a transport spacecraft launched from the earth towards the Lagrangian point carries an assembling laboratory. The autonomous, pico-class, low-power, and low-weight spacecrafts that will explore the asteroid belt for asteroids with certain scientific characteristics will be assembled in that laboratory. Each spacecraft is equipped with a solar sail, which means it relies primarily on power from the sun, using only tiny thrusters to navigate independently. Also, each spacecraft has onboard computation, artificial intelligence, and heuristics mechanism for control at the individual and team levels, and it has communicating mechanism for the communication within swarm and the data transfer

back to the earth, too. Moreover, approximately 80 percent of the spacecrafts will be workers. The workers will carry a single specialized instrument, such as a magnetometer and an X-ray, gamma-ray, visible/IR, or neutral mass spectrometer, for the collection of a specific type of date from asteroids in the belt [44, 45].

The correct assembly of spacecrafts is crucial to the ANTS mission. We reuse the framework of the robotic assembly system which has been studied in [6, 7, 28], and extend it so that it can be used for the assembly of spacecrafts in ANTS mission. As introduced before, the spacecraft consists of five main parts, namely, power system, navigation system, control system, communication system and specialized instrument (i.e. magnetometer or spectromenter). After the extension of the robotic assembly system framework, the left arm of the robot will be responsible for installing power system, navigation system and control system while the right arm will be in charge of the installation of communication system and the specialized instrument.

The main body of formal specification which describes the way how the left arm and the right arm work in the process of assembling the spacecraft is shown as follows. A complete version of the formal description of the robot system in Z notation is displayed in Appendix C.

LeftArmPick

$\Delta RobotSystem$
$part? : Part$

$tempstack = \langle\rangle \lor head\ tempstack \in \mathrm{dom}\ currentproduct \lor$
$head\ tempstack \notin \{PowerSys, NavigationSys, ControlSys\}$
$part? \in \{PowerSys, NavigationSys, ControlSys\}$
$leftarm = \langle\rangle \land leftarm' = \langle part?\rangle$
$tempstack' = tempstack \land rightarm' = rightarm$
$currentproduct' = currentproduct$

LeftArmGetFromStack

$\Delta RobotSystem$

$\#tempstack > 0 \land \#leftarm = 0$
$head\ tempstack \notin \mathrm{dom}\ currentproduct$
$head\ tempstack \in \{PowerSys, NavigationSys, ControlSys\}$
$leftarm' = \langle head\ tempstack\rangle \land rightarm' = rightarm$
$currentproduct' = currentproduct \land tempstack' = tail\ tempstack$

LeftArmRelease

$\Delta RobotSystem$
$part! : Part$

$\#leftarm = 1 \land leftarm(1) \notin \mathrm{dom}\ currentproduct$
$part! = leftarm(1) \land leftarm' = \langle\rangle$
$leftarm(1) = PowerSys \Rightarrow$
$\qquad currentproduct' = currentproduct \cup \{part! \mapsto 1\}$
$leftarm(1) = NavigationSys \Rightarrow$
$\qquad currentproduct' = currentproduct \cup \{part! \mapsto 2\}$
$leftarm(1) = ControlSys \Rightarrow$
$\qquad currentproduct' = currentproduct \cup \{part! \mapsto 3\}$
$tempstack' = tempstack \land rightarm' = rightarm$

LeftArmPushToStack

$\Delta RobotSystem$
$parttopush! : Part$

$\#leftarm = 1 \land leftarm(1) \in \mathrm{dom}\ currentproduct$
$parttopush! = leftarm(1) \land leftarm' = \langle\rangle$
$tempstack' = \langle parttopush!\rangle \frown tempstack$
$rightarm' = rightarm \land currentproduct' = currentproduct$

RightArmPick

$$\Delta RobotSystem$$
$$part? : Part$$

$$tempstack = \langle\rangle \lor head\ tempstack \in \mathrm{dom}\ currentproduct \lor$$
$$head\ tempstack \notin \{CommunicationSys, SpectroMeter, MagnetoMeter\} \lor$$
$$\mathrm{dom}(currentproduct \rhd \{5\}) \cup \{head\ tempstack\} = \{SpectroMeter, MagnetoMeter\}$$
$$part? \in \{CommunicationSys, SpectroMeter, MagnetoMeter\}$$
$$rightarm = \langle\rangle \land rightarm' = \langle part?\rangle \land tempstack' = tempstack$$
$$currentproduct' = currentproduct \land leftarm' = leftarm$$

RightArmGetFromStack

$$\Delta RobotSystem$$

$$\#tempstack > 0 \land \#rightarm = 0$$
$$head\ tempstack \notin \mathrm{dom}\ currentproduct$$
$$\mathrm{dom}(currentproduct \rhd \{5\}) \cup \{head\ tempstack\} \neq \{SpectroMeter, MagnetoMeter\}$$
$$head\ tempstack \in \{CommunicationSys, SpectroMeter, MagnetoMeter\}$$
$$rightarm' = \langle head\ tempstack\rangle \land leftarm' = leftarm$$
$$currentproduct' = currentproduct \land tempstack' = tail\ tempstack$$

RightArmRelease

$$\Delta RobotSystem$$
$$part! : Part$$

$$\#rightarm = 1 \land rightarm(1) \notin \mathrm{dom}\ currentproduct$$
$$\mathrm{dom}(currentproduct \rhd \{5\}) \cup \{rightarm(1)\} \neq \{SpectroMeter, MagnetoMeter\}$$
$$part! = rightarm(1)$$
$$rightarm(1) = CommunicationSys \Rightarrow currentproduct' = currentproduct \cup \{part! \mapsto 4\}$$
$$(rightarm(1) = MagnetoMeter \lor rightarm(1) = SpectroMeter) \Rightarrow currentproduct' = currentproduct \cup \{part! \mapsto 5\}$$
$$rightarm' = \langle\rangle \land tempstack' = tempstack \land leftarm' = leftarm$$

RightArmPushToStack

$$\Delta RobotSystem$$
$$parttopush! : Part$$

$$\#rightarm = 1 \land (rightarm(1) \in \mathrm{dom}\ currentproduct \lor \mathrm{dom}(currentproduct \rhd \{5\}) \cup \{rightarm(1)\} = \{SpectroMeter, MagnetoMeter\})$$
$$parttopush! = rightarm(1) \land rightarm' = \langle\rangle$$
$$tempstack' = \langle parttopush!\rangle \frown tempstack$$
$$leftarm' = leftarm \land currentproduct' = currentproduct$$

The Java program displayed in Appendix D is supposed to implement the robot system. In order to check whether the Java code implements the formal specification correctly, we load both of them to our specification-based monitoring system. As shown in Figure 4.4, the system start monitoring with a sequence of methods i.e. *leftArmPick("PowerSys"); leftArmRelease(); leftArmPick("ControlSys"); leftArmRelease(); rightArmPick("MagnetoMeter"); rightArmRelease(); rightArmPick("CommunicationSys"); rightArmRelease(); rightArmPick("SpectroMeter"); rightArmRelease(); leftArmPick("NavigationSys"); leftArmRelease();*. After method *rightArmRelease()* is executed for the second time, which installs the communication system to the spacecraft, the monitoring system detects an inconformity and reports to the user that operation schema *RightArmRelease* is not implemented correctly.

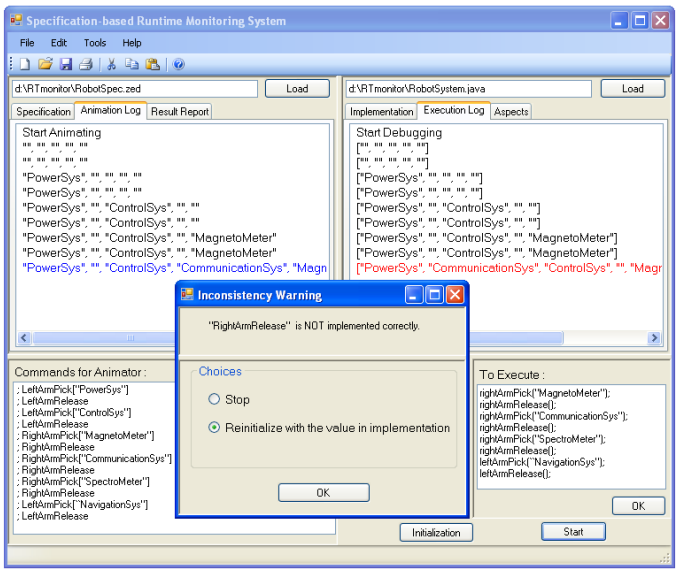

Figure 4.4: Monitor a robotic assembly system

The operation schema, *RightArmRelease*, which is the counterpart of method *rightArmRelease()* in the formal specification and specifies how the right arm will

install components to the spacecraft, is shown as follows.

$$
\begin{array}{|l}
\hline
\textit{RightArmRelease} \\
\Delta \textit{RobotSystem} \\
\textit{part}! : \textit{Part} \\
\hline
\#\textit{rightarm} = 1 \land \textit{rightarm}(1) \notin \operatorname{dom} \textit{currentproduct} \\
\operatorname{dom}(\textit{currentproduct} \rhd \{5\}) \cup \{\textit{rightarm}(1)\} \neq \\
\qquad \{\textit{SpectroMeter}, \textit{MagnetoMeter}\} \\
\textit{part}! = \textit{rightarm}(1) \\
\textit{rightarm}(1) = \textit{CommunicationSys} \Rightarrow \\
\qquad \textit{currentproduct}' = \textit{currentproduct} \cup \{\textit{part}! \mapsto 4\} \\
(\textit{rightarm}(1) = \textit{MagnetoMeter} \lor \textit{rightarm}(1) = \textit{SpectroMeter}) \Rightarrow \\
\qquad \textit{currentproduct}' = \textit{currentproduct} \cup \{\textit{part}! \mapsto 5\} \\
\textit{rightarm}' = \langle\rangle \land \textit{tempstack}' = \textit{tempstack} \land \textit{leftarm}' = \textit{leftarm} \\
\hline
\end{array}
$$

According to the semantics of Z notation, the operation schema *RightArmRelease* specifies that if the right arm is holding an item and the held item is not identical to any existing component of the spacecraft-to-be, the right arm will install the item to a designate position, for example, communication system to the fourth socket.

When we check the Java code in Appendix D, it is found out that method *rightArmRelease()* as shown follows, which corresponds to the operation schema *RightArmRelease*, actually puts the communication system to the second socket. Thus, an error in the program has been localized.

```
public void rightArmRelease() {
    if(!rightarm.isEmpty())
    {   Object releasedPart = rightarm.get(0);
        if (!currentproduct.contains(releasedPart))
        {   if(releasedPart == "CommunicationSys")
            { currentproduct.setElementAt(releasedPart, 1);}
            else if (releasedPart == "MagnetoMeter"||
                                releasedPart == "SpectroMeter")
                {currentproduct.setElementAt(releasedPart, 4);}
            rightarm.clear();
        }
    }
}
```

If we choose to continue the monitoring, when the method *rightArmRelease()* is executed for the third time installing a spectrometer to spacecraft, the monitoring system detects another inconformity. To localize the cause of the inconformity, we revisit the formal specification and the Java code. The operation schema *RightArmRelease* specifies that only one specialized instrument (i.e. magnetometer or spectrometer) will be installed and there is no way to replace it with another one if a specialized instrument has been included in the spacecraft. However, in the Java code, there is no statement for checking whether a specialized instrument has been included in the spacecraft when the right arm tries to install a specialized instrument to the spacecraft. Thereby, another error in the program has been figured out. Furthermore, when the monitoring system proceeds with the methods sequence *leftArmPick("NavigationSys"); leftArmRelease();* the error in the method *leftArmRelease()* will be dug out too.

4.4 Conclusion

This chapter demonstrates the application and effectiveness of the formal specification-based monitoring system with case studies. It has been manifested that the formal specification-based monitoring system can dynamically and continuously checks the conformance of concrete implementations to formal specifications, explicitly recognizes undesirable behaviors in the target system, and responds appropriately in a timely manner as the target system runs.

Chapter 5

AOP-aided Software Evolution

In this chapter, an AOP-aided approach is proposed to handle the evolution of core classes in object-oriented programs when the formal specification of a system has changed. Moreover, the formal specification-based monitoring system presented in Chapter 3 is extended to address the validation of aspect-oriented programs.

5.1 Introduction

After the software has been deployed, new requirements may emerge; the running environment may change; and the performance or reliability may have to be improved. Accordingly, the specification of the system will change. Even in the process of software development, the events that are mentioned above might well happen too. In order to keep conformance with the specification and satisfy the system requirements, changes have to be made to the implementation of the system. This is called software evolution. Software evolution is widely recognized as one of the most important problems among software development technologies, and it is an inevitable and critical stage in the life cycle of any type of software systems, particularly, those serving highly volatile business domains such as banking and telecommunications [74].

Aspect-oriented programming (AOP) has been proposed as a new methodology and a complement to traditional procedural and object-oriented programming (OOP) to improve the separation of concerns in software systems [53, 64, 100]. Traditionally, the focus of AOP is to identify cross-cutting behavior and develop appropriate code that can be added to the base program to realize the cross-cutting behavior. Actually, beyond the improvement of concerns separation, AOP also provides a model to modify a software system after it has been released and installed, which greatly eases the maintenance and evolution of software systems. Therefore, we try to exploit this model and the infrastructure that has been developed for AOP technique for software evolution.

In this chapter, we investigate how aspect-oriented programming technique can help handle the evolution of core classes in object-oriented programs, when the formal specification of a system has changed. We consider two contributions of AOP as important for the evolution of software systems: first, weaving allows an application to be modified at runtime; second, pointcuts provide a mechanism to specify where the code modifications must be applied. Based on them, we proposed an AOP-aided evolution approach. In the proposed approach, the original and new versions of the formal specification of a software system are compared to determine the difference between the two versions of the specification and to identify the changes made to the specification. Then, aspects are constructed to achieve the expected modifications. After the constructed aspects are woven with original classes, the required modification to the implementation will be accomplished.

Furthermore, except the benefits, the introduction of AOP also brings some challenges to software development and maintenance. One of them is the validation of aspect-oriented software systems. The techniques for validating aspect-oriented systems are far behind expectations. In the latter part of the chapter, we extend the formal specification-based monitoring system presented in Chapter 3 so that it can work with aspect-oriented programs. By doing this, we not only provide an approach for aspect-oriented program validation, but also illustrate that our formal specification-based monitoring technique can contribute to the regression test of software systems after evolution has been achieved.

The example presented in this chapter is formally specified with TCOZ notation,

and implemented in Java/AspectJ programming language, but the concepts presented in this chapter are not tied to any specific formal specification language or programming language and can be applied in other contexts as well.

The remainder of the chapter is organized as follows. Section 5.2 introduces AOP and AspectJ briefly. Section 5.3 presents an AOP-aided evolution approach that handles the evolution of core classes in object-oriented programs, when the formal specification of the system has changed. Section 5.4 discuss the feasibility of applying the formal specification-based monitoring technique proposed in Chapter 3 to aspect-oriented programs. Section 5.5 illustrates the proposed evolution approach and validates the resulting aspect-oriented program with the formal specification-based monitoring technique. Section 5.6 concludes this chapter.

5.2 AOP and AspectJ

5.2.1 AOP

The essential idea of aspect-oriented programming (AOP) is that all concerns should be treated as modular units regardless of the limitations of the implementation languages. The primary mechanism for defining solutions to cross-cutting concerns is the *aspect*. Aspects encapsulate behaviors and states of those crosscutting concerns whose implementations must span across the core concerns that form the subject matter of a system. By placing these crosscutting concerns separately in an aspect, the core concerns are made more cohesive since their implementations are relieved of the burden of managing concepts unrelated to their purpose. The

effective modular decomposition facilitates the development of complex systems and makes it easier to understand, maintain and evolve the systems developed.

5.2.2 AspectJ

AspectJ [1, 52, 57] is an implementation of aspect-oriented programming methodology. As a simple and practical aspect-oriented extension to Java, it adds to Java some new concepts and associated constructs such as join points, pointcuts, advice, inter-type declarations and aspects.

Join point in AspectJ is an essential concept in the composition of an aspect with other classes. It is a well-defined point in the execution of a program, such as a call to a method, an access to attribute, an object initialization, or an exception handler.

Pointcut is a set of points that optionally expose some of the values in the execution of the join points. AspectJ defines several primitive *pointcut designators* that can identify all types of join points. Pointcuts in AspectJ can be composed and new pointcut designators can be defined according to these combinations.

Advice is a method-like mechanism used to define certain code that executes *before*, *after*, or *around* a pointcut. The AROUND advice execute in place of the indicated pointcut, which allows the aspect to replace a method. An aspect can also use an *inter-type declaration* to add a public or private method, field, or interface implementation declaration into a class.

Inter-type declaration is a static crosscutting instruction that introduces changes

to classes, interfaces and aspects of the system. It provides definitions of fields, methods, and constructors which makes static changes to the target modules that do not directly affect their behavior but support the implementation of dynamic crosscutting.

Aspects are modular units of crosscutting implementation. In AOP, aspects are the primary mechanism for defining solutions to cross-cutting concerns, which encapsulate behaviors and states of those crosscutting concerns. Aspects are defined by aspect declarations, which have similar forms of class declarations. Aspect declarations may include pointcut, advice, and inter-type declarations, as well as method declarations that are permitted in class declarations.

5.3 AOP-aided Software Evolution

AOP provides the basis for an effective software evolution approach. The mechanism of weaving allows a program to be modified at runtime; moreover, pointcuts provide a mechanism to specify where the code modifications must be applied. Therefore, crosscutting concerns that correspond to the required changes can be added to original programs without making any invasive modifications.

In this section, we present an AOP-aided software evolution approach that handles the evolution of core classes in object-oriented programs when the formal specification of the system has changed. In this chapter, we assume the system is specified in TCOZ notation and implemented in Java/AspectJ programming language.

5.3.1 Determine Differences

First of all, the changes to the original system specification, which is corresponding to the changes of the system requirement, should be identified.

Given the new and original versions of the formal specification of a system, there is a function which figures out the differences between two versions and outputs the sets of added members (state variables/fields and operations/methods), removed members, and modified members. The algorithm behind the function is shown in Figure 5.1.

Taking the two versions of the formal specifications as inputs, the function first figures out the set of the state variables that are common to the original and new versions, *ComnSV* (`Line 1`); the set of the state variables that are newly added to the specifications, *AddSV* (`Line 2`); and the set of the state variables that are removed from the specifications, *RmvSV* (`Line 3`). For every state variable that is common to the two versions of the formal specification, if the type of the state variable is different in the two version of specifications, it is identified as a modified variable and put into the set of modified state variables, *MdfSV* (`Line 4-9`). Then, the function figures out the set of the operations that are common to the original and new versions, *ComnOP* (`Line 10`); the set of the operations that are newly added to the specifications, *AddOP* (`Line 11`); and the set of the operations that are removed from the specifications, *RmvOP* (`Line 12`). For every operation that is common to the two versions of the formal specification, if the operation schema or CSP process that is used to defined this operation is different in the two version

of specifications, the operation is identified as a modified operation and put into the set of modified operations, *MdfOP* (`Line 13-18`).

Algorithm SpecDiff (OldSpec, NewSpec): AddSV, RmvSV,MdfSV, AddOP, RmvOP, MdfOP

input:

OldSpec : old version of the system's formal specification
NewSpec : new version of the system's formal specification

output:

AddSV: the set of added state variables
RmvSV: the set of removed state variables
MdfSV: the set of modified state variables
AddOP: the set of added operations
RmvOP: the set of removed operations
MdfOP: the set of modified operations

global:

ComnSV: the set of state variables that are common to the original and new versions of the specification
ComnOP: the set of operations that are common to the original and new versions of the specification

```
1.  ComnSV ⟵ OldSpec.StateVariables ∩ NewSpec.StateVarialbes
2.  AddSV ⟵ NewSpec.StateVariables \ ComnSV
3.  RmvSV ⟵ OldSpec.StateVariables \ ComnSV
4.  for each Sv ∈ ComnSV do
5.    opdiff ⟵ Compare(NewSpec.Sv.Type, OldSpec.Sv.Type)
6.    if opdiff then
7.        MdfSV ⟵ MdfSv ∪ {NewSpec.SV}
8.    end if
9.  end for
10.  ComnOP ⟵ OldSpec.OP ∩ NewSpec.OP
11.  AddOP ⟵ NewSpec.OP \ ComnOP
12.  RmvOP ⟵ OldSpec.OP \ ComnOP
13.  for each Op ∈ ComnOP do
14.  opdiff ⟵ Compare(NewSpec.Op, OldSpec.Op)
15.    if opdiff then
16.        MdfOP ⟵ MdfOP ∪ {NewSpec.Op}
17.    end if
18.  end for
```

Figure 5.1: Difference determination

5.3.2 Construct Aspects

After the difference between the original and new versions of the formal specification has been identified, we can construct the aspects that would modify the original system without invasive modifications on it. The approach to construct the aspect is described in the algorithm shown in Figure 5.2.

Algorithm CtrAspect (AddSV, MdfSV, AddOP, MdfOP): NewAspect
input:
AddSV: the set of added state variables
MdfSV: the set of modified state variables
AddOP: the set of added operations
MdfOP: the set of modified operations
output:
NewAspect: an aspect crosscutting the original class

1. declare *NewAspect*, an aspect crosscutting the original class
2. **for each** Sv ∈ AddSV ∪ MdfSV **do**
3. inter-type declaration *Dlr_sv* is added to *NewAspect*
4. **end for**
5. **for each** Op ∈ AddOP **do**
6. inter-type declaration *Dlr_op* is added to *NewAspect*
7. **end for**
8. **for each** Op ∈ MdfOP **do**
9. pointcut *PC_op*: *call(Op)* is added to *NewAspect*
10. around-advice *AD_op* is added to *NewAspect*
11. **end for**

Figure 5.2: Aspect construction

With the information about the difference between the original specification and the new specification as input, the algorithm will construct an aspect, *NewAspect*, which will crosscut the original class (`Line 1`). For each new state variable, an inter-type declaration *Dlr_sv*, which declares and initializes a new field and targets

the original class, will be added to *NewAspect*. Meanwhile, the modification that can be made to the state variable is the change of type. We can take it as an introduction of a new state variable. Therefore, for each modified state variable, an inter-type declaration *Dlr_sv*, which declare the corresponding class variable with the new type and initializes it, is added to *NewAspect* (`Line 2-4`). For each new operation, an inter-type declaration *Dlr_op*, which declares a method corresponding to the new operation, is introduced to *NewAspect* (`Line 5-7`). For each modified operation, *PC_op*, a pointcut which captures all the calls of the corresponding method is constructed; meanwhile, *AD_op*, an *around*-advice which implements the functions described by the modified operation is constructed and bound with the pointcut *PC_op* (`Line 8-11`).

5.3.3 Weave Constructed Aspect with Original Class

After construct the aspect, we remove class variables corresponding to state variables that are removed and modified in the new version of the specification from the original implementation. We also remove the methods corresponding to operations that are removed in the new version of the specification, using the weaving mechanism provided by aspect-oriented programming technique. Then, the aspect we have constructed will be woven with the original implementation. As a result, a new version of the implementation can be generated and an AOP-aided evolution is achieved.

5.4 Monitor Aspect-oriented Programs

On the one hand, AOP is proposed as a new methodology and a complement to traditional object-oriented programming (OOP) to improve the separation of concerns in software systems. The effective modular decomposition facilitates the development of complex systems and makes it easier to understand, maintain and evolve the systems developed.

On the other hand, just as with the introduction of OOP, AOP not only brings a unique set of benefits, such as high modularity and low maintenance burden; but also introduces a set of challenges, such as new problems with respect to the verification and test of the systems developed with AOP. By far, research in AOSD has focused mostly on the activities of problem analysis and language implementation. The techniques for validation and verification are still far behind expectation and what has been achieved for the static analysis of procedural and object-oriented programs, although they are being developed for aspect-oriented systems. Because the expressive power that aspects unleash heightens the potential for insidious errors, finding cost-effective verification techniques that address aspect-oriented software is especially important to AOP.

In Chapter 3, we have proposed a formal specification-based monitoring approach. In the proposed technique, the formal specification of a system is animated to exhibit the expected behaviorial properties of the target system. The valuable information about desired dynamic behaviors of the system is extracted from the

animation. And, the information about dynamic behaviors of concrete implementations of the target system is obtained through program debugging. Base on the attained information from both sides, the judgement on the conformance of the concrete implementation with the formal specification is timely made when the system is executed.

We illustrated the capability of the monitoring technique with object-oriented programs in Chapter 3. The underlying idea can work with aspect-oriented programs too. With the runtime monitoring technique, we will be capable of checking whether the software system resulting from the AOP-aided software evolution technique behaves as expected by the system requirements which are described in formal specification notations.

We have integrated the mechanism for animating TCOZ specification developed by Sun *et al.* [97] into the prototype monitoring system and extended the monitoring system so that it can work with aspect-oriented programs. We will illustrate the monitoring of an aspect-oriented program with an example in the next section.

5.5 A Case Study

In this section, we demonstrate the proposed AOP-aided evolution approach and runtime monitoring technique with a case study.

We take the *Timed Message Queue System* introduced in Chapter 2 as an example and customized the type $[MSG]$ as *natural number*. The formal specification is redisplayed in Figure 5.3.

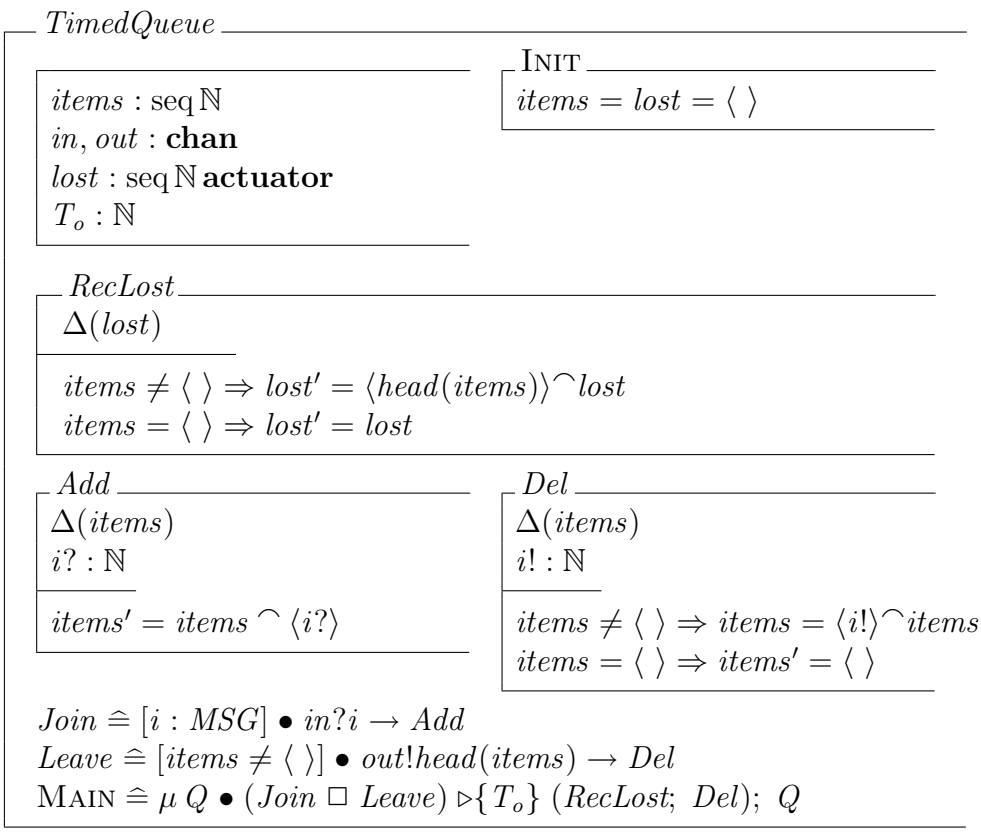

Figure 5.3: *TimedQueue* in TCOZ notation

The TCOZ specification in Figure 5.4 describes a queue which has evolved from what is described in Figure 5.3. They are similar except that, in the new version, a new state variable *ineligibleElement* is introduced and there is an *ineligible element rule* which demands that any element added to the *Timed Message Queue System* should not equal to *ineligibleElement*.

While the difference between the original and new versions of the specification has been identified, an aspect is constructed following the algorithm in Figure 5.2. As shown in Figure 5.5, the *IneligibleElementRuleAspect* aspect introduces a data member *_ineligibleElement* of type *Object* into the original *Queue* class. Mean-

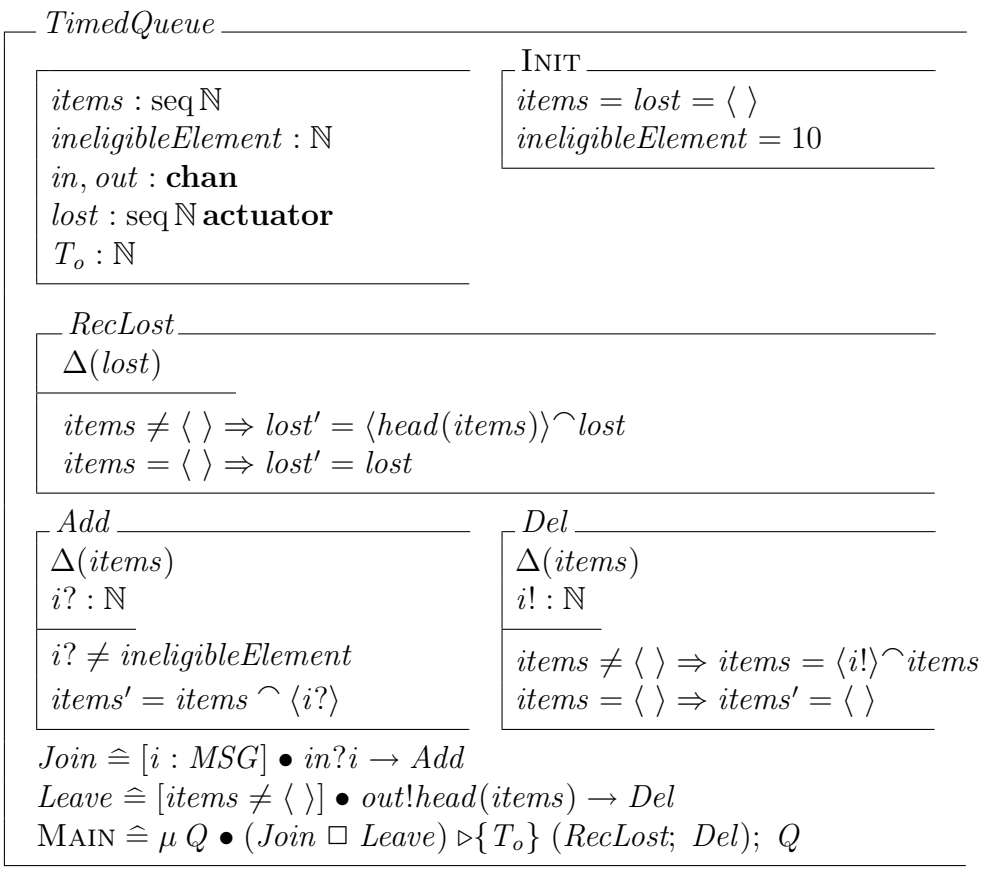

Figure 5.4: Modified *TimedQueue* in TCOZ notation

while, through two pieces of advice, the *IneligibleElementRuleAspect* aspect modifies the execution behavior of the original *Queue* class in the following way: firstly, whenever a new object of *Queue* class is created, the date member *_ineligibleElement* will be initialized as 10; secondly, before a new item is added to the queue, check whether it is an eligible element for the queue; if not, then nothing will be added to the queue and an exception will be thrown. The *IneligibleElementRuleAspect* aspect implements the *ineligible element rule* and cuts across the class which is a correct implementation of the specification displayed in Figure 5.3. After weaving the *IneligibleElementRuleAspect* aspect with the original *Queue* class, the evolution

```
public aspect IneligibleElementRuleAspect {
 private Object Queue._ineligibleElement;
 after(Queue queue):
   execution(Queue.new(..)) && this(queue) {
   queue._ineligibleElement = "10";
  }
 before(Queue queue, Object item)
  throws IneligibleElementException :
  execution(* Queue.enQueue(*))&& this(queue) && args(item) {
    if (item == queue._ineligibleElement) {
       throw new IneligibleElementException(
          "This item is ineligible for this queue");
    }
  }
}
```

Figure 5.5: Aspect for ineligible element rule

required by the new version of specification is supposed to achieve.

Now, we try to verify whether the program achieved through the AOP-aid evolution behaves as expected by the new version specification using our monitoring technique. After load the monitoring system with the new version of specification, the class *Queue* and the aspect *IneligibleElementRuleAspect* and finish configuration, we input the sequence of methods: enQueue("5"); enQueue("15"); enQueue("10"), and enable the monitoring system to work. When the method enQueue("10") is executed, the monitoring system reports an inconformity as show in Figure 5.6, informing the user that the operation *Add* is not implemented correctly, and provides the user with two choices for what to do next. Checking the AspectJ code in Figure 5.5, we can find out that the last advice in aspect *IneligibleElementRuleAspect* only throws an exception but not prevents the operation that adds an item into the queue from being invoked. To fix this problem, the advice should be changed to

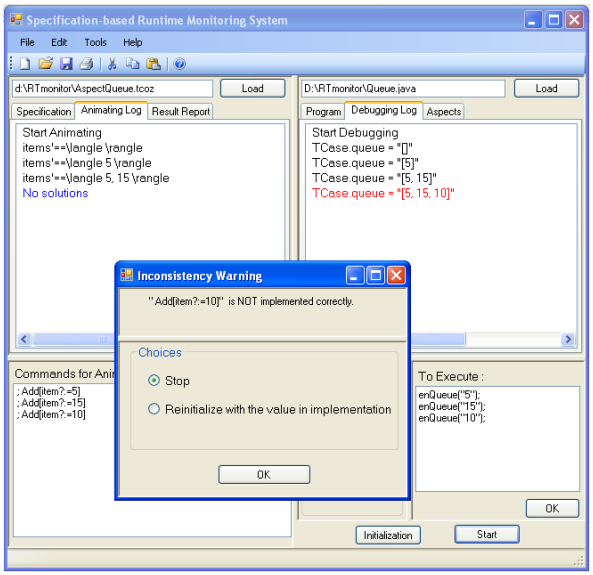

Figure 5.6: Monitoring aspect-oriented program

```
void around(Queue queue, Object item):
     execution(* Queue.enQueue(*))
           && this(queue) && args(item) {
       if (item.euqals(queue._ineligibleElement)) {}
           else{proceed(account, amount);}
```

Figure 5.7: Correct advice

the one shown in Figure 5.7. With the around advice, the aspect *IneligibleElementRuleAspect* modifies the execution behavior of the *Queue* module in a way that an item is not allowed to be added into the queue if it equals the *_ineligibleElement*. This is exactly what is described by the new version specification shown in Figure 5.4. With the correct aspect loaded to the monitoring system, no inconformity is reported. It reassures the user that, after woven, the aspect *IneligibleElementRuleAspect* and the class *Queue* implement the specification as expected.

5.6 Conclusion

This chapter presents an AOP-aided evolution approach that can handle the evolution of core classes in object-oriented programs, when the formal specification of the system has changed. In the proposed approach, the original and new versions of the formal specification of a software system are compared to determine the difference between the two versions of the specification and to identify the changes made to the specification. Then, aspects are constructed to achieve the expected modifications. After the constructed aspects are woven with original classes, the required modification to the implementation will be accomplished.

Furthermore, the formal specification-based monitoring system presented in Chapter 3 is extended to work with aspect-oriented programs. By doing this, we not only provide an approach for aspect-oriented program validation, but also illustrate that our formal specification-based monitoring technique can contribute to the regression test of software systems after evolution has been achieved.

Chapter 6

Formal Specification-based Regression Test Suite Construction

In this chapter, a formal specification-based regression test suite construction technique is presented. It addresses both the *regression test selection problem* and the *test suite augmentation problem*, is presented.

6.1 Introduction

After the software has evolved according to the changes of the specification or requirement of the software system, regression testing has to be performed on the modified software to provide the confidence that the changed parts of the software behave as intended and that the unchanged parts have not been adversely affected by the modifications.

In order to reduce the cost and effort required by regression testing, the test suite, which was used to test the original version of the software, are reused for regression testing. However, rerunning all test cases in the original test suite can be prohibitively expensive; and some test cases in the original test suites may be obsolete to the modified version of the system. Therefore, regression test selection techniques are required for efficient regression testing. Most of the existing regression test techniques [9, 19, 59, 84, 106, 110, 48, 85, 109, 37] are strictly code-based. They select test cases for the regression testing, from the original test suites, only using the information gathered by code analysis.

However, if the specifications for the software have been changed while code modifications necessary to implement the changed specifications have not been made, such a fault can only be detected by specification-based test case selection which selects the test cases related to the changed specifications [85]. Thereby, when selecting test cases for regression testing, specification-based techniques are required as well as code-based techniques.

Furthermore, new test cases might well be required in regression testing because of the changes made to the software systems or the new functionalities introduced to the systems. This is the *test suite augmentation problem* [37]. With the solid mathematical bases, formal specification provides a good starting point to address the test suite augmentation problem, and to systematically generate new test cases that are required for the testing of new functionalities or certain changed parts of software systems.

In this chapter, a formal specification-based regression test suite construction technique, which works with the formal specifications written in Timed Communicating Object-Z (TCOZ) notation [66, 70], is proposed. The proposed technique addresses not only the *regression test selection problem* but also the *test suite augmentation problem* for the regression testing of classes in object-oriented programs. It first constructs the two versions of control flow representations for the classes, according to the original and changed versions of the classes's formal specification. Then, it compares the original control flow representation and the original formal specification with their respective modified version to figure out the modifications made to the specification. And, sequentially, it uses the information about the modifications made to the specification to identify the obsolete test cases, to select the test cases related to the changed specifications from the original test suite, and to guide the generation of new test cases for regression testing.

The proposed regression test suite construction technique is strictly specification-based. No complex static or dynamic code analysis is required for it to work.

Therefore, it is independent of the programming language that is used to implement the classes described by the formal specifications and it can be automated. In the latter part of this chapter, a TCOZ specification-based regression test suite construction system named TcozRts, which implements our formal specification-based technique for the regression test suite construction, is presented.

The rest of the chapter is organized as follows. In the next section, the distinct characteristics of the way in which classes are specified in TCOZ notation are discussed. In Section 6.3, a graphic representation for the class which is specified in TCOZ notation is proposed. The formal specification-based regression test suite construction technique for the class specified in TCOZ notation is presented in Section 6.4. In Section 6.5, a TCOZ specification-based regression test suite construction system is described in details. Finally, Section 6.6 concludes this chapter.

6.2 Classes Specified in TCOZ Notation

As introduced in Chapter 2, the basic structure of a TCOZ document is the same as that of an Object-Z document, consisting of a sequence of class definitions, and some type and constant definitions in the usual Z style. However, TCOZ varies remarkably from Object-Z in the definition of class schemas.

Firstly, in TCOZ notation, operation schemas (both syntactically and semantically) is identified with (terminating) CSP processes that perform only state update events. Active classes are identified with non-terminating CSP processes; and the MAIN process determines the behaviour of the objects of an active class after

initialization. Therefore, in TCOZ, operation schemas and CSP processes occupy the same syntactic and semantic category. It means that operation schema expressions may appear wherever processes may appear in CSP and CSP process definitions may appear wherever operation definitions may appear in Object-Z. In fact, all operation definitions in TCOZ are considered as the definitions CSP processes. Furthermore, it is natural to allow to define TCOZ operations in terms of CSP primitives as well as through the schema calculus. By allowing an operation to consist of a number of events, it becomes feasible to specify its temporal properties when describing the operation. Meanwhile, operation schemas take on the syntactic role of CSP processes, so they may be combined with other schemas and even CSP processes using the standard CSP process operators. Thus it becomes possible to represent true multi-threaded computation even at the operation level.

Secondly, in TCOZ notation, the class state-schema convention is extended to allow the declaration of communication channels. If c is to be used as a communication channel by any of the operations of a class, then it must be declared in the state schema to be of type **chan**. Channels are type heterogeneous and may carry communications of any type. Contrary to the conventions adopted for internal state variables, channels are viewed as global rather than as encapsulated entities. This is an essential consequence of their roles as communications interfaces *between* objects. The introduction of channels to TCOZ reduces the need to reference other classes in class definitions, thereby further enhancing the modularity of system specifications.

Lastly, in TCOZ notation, the state-guard which is a CSP operator is used to *block* or *enable* execution of an operation on the basis of an object's local state. For example, the operation $[a \geq 0] \bullet [\Delta(a) \mid a \geq 0 \land a' = \sqrt{a}\,]$ will replace the state variable a with its square root if a is positive otherwise it will *deadlock*, that is be blocked from executing. The blocking or enabling of this operation is achieved by the state guard $[a \geq 0] \bullet \_$ and not by the precondition $a \geq 0$ within the operation schema. If the operation schema alone is invoked with a negative, it will *diverge* rather than block. The difference between deadlock and divergence is that a divergence may be refined away by making an operation more robust, while a deadlock can never be refined away. An additional function of state guards is as a substitute for CSP's indexed external choice operator. The process $[n : \mathbb{N} \mid 0 \leq n \leq 5] \bullet c?n \rightarrow P(n)$ may input any value of n between 0 and 5 (from channel c) as chosen by its environment. CSP's indexed internal choice is replaced by the operation schema and sequential composition. The process $[n! : \mathbb{N} \mid 0 \leq n! \leq 5];\ c!n \rightarrow P(n)$ may output any value of n between 0 and 5 according to its own designs.

6.3 Representation for Classes Specified in TCOZ

The characteristics discussed in the previous section make it possible to construct a graphic representation for the classes that are specified in TCOZ notation, according to the semantics of TCOZ. We develop such a graphic representation and name it *testchart*. In this section, an overview of testchart is presented; and the

interaction coverage criterion, which is proposed based on the testchart and will be used in our regression testing technique, is also introduced.

6.3.1 Testchart

The *testchart* for a class depicts all the possible interactions among the operations of a class and the order in which the operations can be invoked, just like a control flow graph of program statically represents all possible program paths. It is developed for TCOZ specification-based class testing where it is used for test case generation and coverage evaluation. In regression testing, the testchart serves as the base for detecting modifications made to the specification of a class. By representing a class with a testchart, the original and modified versions of the testchart can be compared to reveal the modifications made to the control flow structure of the specification of a class and the information about the modifications will be used to identify the obsolete test cases and to guide the generation of new test cases for regression testing.

Formally, a *testchart* for a class is a 3-tuple $\langle S, T, s_0 \rangle$ where

1. S is a finite, non-empty set of vertices. Each vertex in S represents an operation of the class. Each of the operations that appear in the definition of process MAIN, MAIN itself included, has a corresponding vertex in the testchart.

2. T $\subseteq S \times S$ is a set of directed edges between vertices. An edge (A, B) $\in$ T if and only if it is permissible for a client module to invoke the operation

represented by A followed by operation represented by B.

3. $s_0 \in$ S is the vertex that represents the *Init* operation of the class.

The state-guards in TCOZ specification [67] are used to label the corresponding edges in the testchart.

$[MSG]$ [messages]

TimedQueue

$items : \text{seq}\, MSG$
$in, out : \textbf{chan}$
$lost : \text{seq}\, MSG\ \textbf{actuator}$
$T_o : \mathbb{N}$

INIT
$items = lost = \langle\ \rangle$

RecLost
$\Delta(lost)$

$items \neq \langle\ \rangle \Rightarrow lost' = \langle head(items) \rangle \frown lost$
$items = \langle\ \rangle \Rightarrow lost' = lost$

Add
$\Delta(items)$
$i? : MSG$

$items' = items \frown \langle i? \rangle$

Del
$\Delta(items)$
$i! : MSG$

$items \neq \langle\ \rangle \Rightarrow items = \langle i! \rangle \frown items'$
$items = \langle\ \rangle \Rightarrow items' = \langle\ \rangle$

$Join \mathrel{\widehat=} [i : MSG] \bullet in?i \rightarrow Add$
$Leave \mathrel{\widehat=} [items \neq \langle\ \rangle] \bullet out!head(items) \rightarrow Del$
$\text{MAIN} \mathrel{\widehat=} \mu\, Q \bullet (Join \,\Box\, Leave) \rhd \{T_o\}\ (RecLost;\ Del);\ Q$

Figure 6.1: *TimedQueue* in TCOZ notation

In Chapter 2, we introduced a *Timed Message Queue System* to illustrate the way that classes are specified in TCOZ notation. In this chapter, we revisit the *Timed Message Queue System* as shown in Figure 6.1, and take it as an example to show how our formal specification-based regression testing technique works.

Following the definition of testchart and the semantics of TCOZ notation, the testchart constructed for the *Timed Message Queue System* is shown in Figure 6.2.

In the TCOZ specification of the *Timed Message Queue System* shown in Figure 6.1, the MAIN process is defined as follows:

$$\text{MAIN} \mathrel{\widehat=} \mu\, Q \bullet (Join \,\Box\, Leave) \rhd \{T_o\}\ (RecLost;\ Del);\ Q$$

According to the semantics of TCOZ , the system will make a choice between operation *Join* and *Leave* , if neither of them is invoked within a certain time 'T_o' then operation *Reclost* will be invoked followed by operation *Del*. Therefore, in the testchart for the *Timed Message Queue System*, there are directed edges from *Main* to *Join, Leave and RecLost* respectively and also a directed edge from *RecLost* to *Del*. Since MAIN is a recursively-defined process, there are directed edges from *Join, Leave* and *Del* to *Main*. In the definition of operation *Leave*, there is a state guard $[items \neq \langle\rangle]$; so the directed edge from *Main* to *Leave* will be labelled by $\sharp items > 0$ which is a variant of $items \neq \langle\rangle$. As a result, we get the corresponding testchart as Figure 6.2 shows.

6.3.2 Coverage Criteria

Coverage criteria specify the set of elements to be covered in testing and guide the selection of test cases. They provide stopping rules for testing (i.e., the rules to determine whether sufficient testing has been performed and the test can be stopped) and the measurements of test-suite quality (i.e., the degree of adequacy

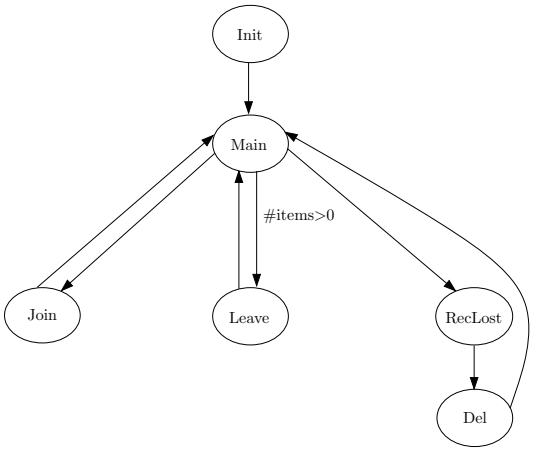

Figure 6.2: Testchart for *TimedQueue*

associated with a test suite) [117].

There have been some well-known coverage criteria for code-based testing, such as *statement coverage*, *condition coverage*, *branch coverage*, *path coverage*, *mutation coverage*, *c-use coverage*, *p-use coverage*, *all-uses coverage* and *du-paths coverage* [12, 47, 81]. Mathur and Wong [71] conducted an empirical study to compare the difficulty and costs of satisfying the data flow based all-uses criterion and a mutation based criterion. They also provided a theoretical comparison between the c-use, p-use, and all-uses coverage criteria and a mutation based criterion in [72]. For specification-based testing, a few coverage criteria have been proposed too. For instance, Offutt and Liu *et al.* [78, 79] defined four criteria at different levels of abstraction on the specifications for the test generation from state-based specifications, namely, transition coverage, full predicate coverage, transition-pair coverage and complete sequence. Andrews and France *et al.* [8] proposed test adequacy criteria, based on UML model elements, for testing executable forms of UML models.

Class testing typically involves the invocation of sequences of operations (methods) in various orders on the objects of the class. The set of test sequences, which satisfies the test requirement for the class-level testing, demands that all the interactions between methods be covered. The directed edges in the testchart correspond to the interactions between the methods specified in TCOZ specifications. Therefore, based on the testchart, we propose *interaction coverage* criterion and we will use it in our regression test technique to guide regression test selection and test cases generation.

Interaction Coverage – A set TS of test sequences satisfies the *interaction coverage* criterion if only if for all directed edges (a, b) $\in$ T, there is at least one test sequence $t \in$ TS such that t contains (a, b).

6.4 Regression Test Suite Construction

In this section, we present a TCOZ specification-based test suite construction technique for the regression testing of classes in object-oriented programs. The presentation begins with an overview of the technique. Next, we itemize the modifications to the formal specification that we will take into consideration in the proposed regression testing technique and discuss the impacts of those modifications on the original test cases. Then, a regression test selection algorithm is presented. Finally, we illustrate the proposed technique with an example.

6.4.1 Overview of the Technique

Our regression test suite construction technique functions on testcharts to select the test cases related to the changed specifications from the existing test suite and to generate new test cases for regression testing. It performs the following five main steps: (1) it parses the original and modified versions of specifications to identify modified operations, (2) it constructs testcharts to represent the original and modified versions of the class' specifications respectively, (3) it compares the two versions of the testchart to identify the added/deleted operations and the added/deleted interactions, (4) based on the information obtained through steps 1 and 3, it identifies the obsolete test cases and selects the test cases related to the changed specifications, from the original test suite, for reuse in the regression testing, (5)based on the information obtained through step 3, it generates test cases for testing the new operations and interactions that are added to the modified version of the specification, so that the interaction coverage criterion will be satisfied in the regression testing of the class.

6.4.2 Modifications to TCOZ Specification

In our TCOZ specification-based regression test suite construction technique, we consider the following kinds of modifications that had been made to the specification of a class.

- **Modified operation**: We assume that an operation has a unique name and is not renamed across the different versions of the class' specification unless

it does't exist in the modified version. In TCOZ, the operations of a class can be defined in the form of operation schema (e.g. *Add* in Figure 6.1) or in the form of CPS process (e.g. *Join* in Figure 6.1). The change of an operation may result from the addition/deletion of an attribute or the change of one of the attributes the class can access. In TCOZ specification, the attributes of a class are declared in the state-schema. An added/deleted attribute is an attribute that is not declared in the original/modified version of the specification of a given class, but is declared in the modified/original version of the specification. A modified attribute is an attribute which exists in both versions of the specification but with different scope, type, or visibility. The change of an operation may also result from the change of a precondition or postcondition when it is defined in the form of operation schema, or the change of the process when it is defined in terms of CSP process. The modified operations can be identified automatically by parsing the TCOZ specification of a class.

- **Added/Deleted operation**: An added/deleted operation is an operation that does not exist in the original/modified version of the class' specification, but exists in the modified/original version of the specification. As stated in the definition of testchart which is shown in Section 6.3, each vertex of a testchart represents an operation of the corresponding class. Thereby, we assume that the function $V(testchart)$ returns the set of vertices for the *testchart* in question; *operations-added* is the set of added operations;

operations-deleted is the set of deleted operations; $Testchart_o$ and $Testchart_m$ are the testcharts for the original version and modified version of a class' specification, respectively. Thus, we can have the following statements.

$$\textbf{operations-added} \leftarrow \mathrm{V}(Testchart_m) \text{ - } \mathrm{V}(Testchart_o)$$

$$\textbf{operations-deleted} \leftarrow \mathrm{V}(Testchart_o) \text{ - } \mathrm{V}(Testchart_m)$$

The relative complement of $\mathrm{V}(Testchart_o)$ relative to $\mathrm{V}(Testchart_m)$ corresponds to the set of operations that are added to the original specification. Meanwhile, the relative complement of $\mathrm{V}(Testchart_m)$ relative to $\mathrm{V}(Testchart_o)$ corresponds to the set of operations that are deleted from the original specification. With the above two statements, the operations that are added to or deleted from the original version of specification can be identified automatically based on the testchart.

- **Added/Deleted interaction**: An added/deleted interaction is an interaction between the operations of a class, which does not exist in the origianl/modified version of the class' specification but exists in the modified/original version of the specification. In TCOZ, active classes are identified with non-terminating CSP processes and the MAIN process determines the behaviour of objects of an active class after initialization. The testchart, which is derived from TCOZ specifications of the class under test as we discussed in Section 6.3, depicts the control flow relationships among the operations of the class. The directed edges in the testchart correspond to the interactions between the operations. Thereby, we assume that the

function $E(testchart)$ returns the set of edges for the testchart in question; *interactions-added* is the set of added interactions; *interactions-deleted* is the set of deleted interactions; $Testchart_o$ is the testchart for the original specification of a class and $Testchart_m$ is the testchart for the modified version of the specification. Thus, we can have the following statements.

$$\textbf{interactions-added} \leftarrow \mathrm{E}(Testchart_m) \text{ - } \mathrm{E}(Testchart_o)$$

$$\textbf{interactions-deleted} \leftarrow \mathrm{E}(Testchart_o) \text{ - } \mathrm{E}(Testchart_m)$$

The relative complement of $\mathrm{E}(Testchart_o)$ relative to $\mathrm{E}(Testchart_m)$ corresponds to the set of interactions that are added to the original specification. Meanwhile, the relative complement of $\mathrm{E}(Testchart_m)$ relative to $\mathrm{E}(Testchart_o)$, corresponds to the set of interaction that are deleted from the original specification. With the above two statements, the interactions that are added to or deleted from the original version of specification can also be identified automatically based on the testchart.

6.4.3 Impacts of Modifications on Test Cases

Following the classification presented by Leung and White [58], we classify original test sequences into three categories: obsolete, retestable and reusable.

- Obsolete: A test sequence is *obsolete* if it is an invalid sequence of operations in the modified version of the class' specification.

- Retestable: A test sequence is *retestable* if it remains valid in the modified version of the class' specification, but one or more of the operations in the sequence have been modified.

- Reusable: A test sequence is *reusable* if it is valid and consists of operations that have remained unchanged in the modified version of the class' specification.

In regression testing, it is necessary to identify the obsolete test sequences and remove them from the original test suite if any test sequence reuse is desired. Meanwhile, the retestable test sequences must be rerun to ensure that the regression testing is safe while the reusable test sequence do not need to be rerun in regression testing.

Furthermore, it should be noted that, in the specification-based regression testing of classes, both the addition/deletion of operations and the addition/deletion of interactions among operations in the specification of the class could make a original test sequence obsolete.

6.4.4 Regression Test Selection Algorithm

Now, a TCOZ specification-based regression test selection algorithm is presented. To identify obsolete test sequences and select test sequences from the original test suite, each test sequence is associated with two bit vectors, *Operations-Used* and *Interactions-Used*. The algorithm is shown in Figure 6.3. It takes a number of parameters: (1) the original test suite **TS**, (2) the set of modified operations **OpMd**,

Algorithm TestSelection(TS, OpMd, OpDel, IntrActDel): RTS

input:
TS: original test suite;
OpMd: the set of modified operations;
OpDel: the set of deleted operations;
IntrActDel: the set of deleted interactions

output:
RTS: subset of TS selected for use in regression testing

global:
Obslt: subset of TS obsolete for regression testing

```
1.    Begin
2.      Obslt = φ    RTS = φ
3.      for each (ts ∈ TS) do
4.        get bit vector Operations-Used of ts
5.        for each (O_i ∈ OpDel)do
6.          while (O_i^th bit of Operations-Used == 1) do
7.            Obslt = Obslt ∪ {ts}
8.          endwhile
9.        endfor
10.       get bit vector Interactions-Used of ts
11.       for each (IntrAct_i ∈ IntrActDel) do
12.         while (IntrAct_i^th bit of Interactions-Used == 1) do
13.           Obslt = Obslt ∪ {ts}
14.         endwhile
15.       endfor
16.     endfor
17.     for each (ts ∈ (TS - Obslt)) do
18.       get bit vector Operations-Used of ts
19.       for each (O_i ∈ OpMd) do
20.         while (O_i^th bit of Operations-Used == 1) do
21.           RTS = RTS ∪ {ts}
22.         endwhile
23.       endfor
24.     endfor
25.   End
```

Figure 6.3: Formal specification-based regression test selection algorithm

(3) the set of deleted operations **OpDel**, and (4) the set of deleted interactions **IntrActDel**. It returns **RTS**, the subset of **TS** which is selected for use in regression testing.

The algorithm starts by initializing **Obslt**, which will hold test sequences that are obsolete for regression testing, to *empty*, and initializing **RTS** to *empty*. For each *ts* from **TS**, the algorithm attains the bit vector *Operations-Used* of *ts* (`Line 4`). If *ts* includes an operation O_i which does not exist in the new version of the specification (`Lines 5, 6`), then *ts* is identified as obsolete (`Line 7`). The algorithm continues by attaining the bit vector *Interactions-Used* of *ts* (`Line 10`). If *ts* includes an interaction $IntrAct_i$ which does not exist in the new version of the specification (`Lines 11, 12`), then *ts* is identified as obsolete (`Line 13`). Having identified all the obsolete test sequences, for each *ts* that is an element of **TS** but not an element of **Obslt**, the algorithm attains the bit vector *Operations-Used* of *ts* (`Line 18`). If *ts* includes an operation O_i which is modified in the new version of the specification (`Lines 19, 20`), then *ts* must be selected for regression testing (`Line 21`).

6.4.5 A Case Study

To demonstrate the formal specification-based regression test suite construction technique, we apply it to the *Timed Message Queue System* that we have introduced before. The specification of the original *Timed Message Queue System* has been shown in Figure 6.1. The specification of a modified version of *Timed Message Queue System* is presented in Figure 6.4. By parsing the two versions of the

$Flag ::= In \mid NotIn$

ModifiedTimedQueue

$items : \text{seq}\, MSG$
$in, out : \textbf{chan}$
$T_o : \mathbb{N}$

INIT
$items = \langle\, \rangle$

Add
$\Delta(items)$
$i? : MSG$

$items' = \langle i? \rangle \frown items \frown \langle i? \rangle$

Find
$i? : MSG$
$flag! : Flag$

$i? \in item \Rightarrow flag! = In$
$i? \notin item \Rightarrow flag! = NotIn$

Del
$\Delta(items)$
$i! : MSG$

$items \neq \langle\, \rangle \Rightarrow items = \langle i! \rangle \frown items'$
$items = \langle\, \rangle \Rightarrow items' = \langle\, \rangle$

$Join \mathrel{\widehat=} [i : MSG] \bullet in?i \rightarrow Add$
$Leave \mathrel{\widehat=} [items \neq \langle\, \rangle] \bullet out!head(items) \rightarrow Del$
$\text{MAIN} \mathrel{\widehat=} \mu\, Q \bullet (Join \,\Box\, Leave \,\Box\, Find) \triangleright \{T_o\}\; Del;\; Q$

Figure 6.4: Modified *TimedQueue*

specification, it can be identified that operation *Add* has been modified. In the modified version of the specification, given an i of type *MSG*, operation *Add* will attach it to both head and tail of the list. *Add* appears in the definition of operation *Join*; the semantics of operation *Join* is modified due to the modification to *Add*. As a result, we obtain the set

$$\textbf{operations-modified} = \{\text{Add, Join}\}$$

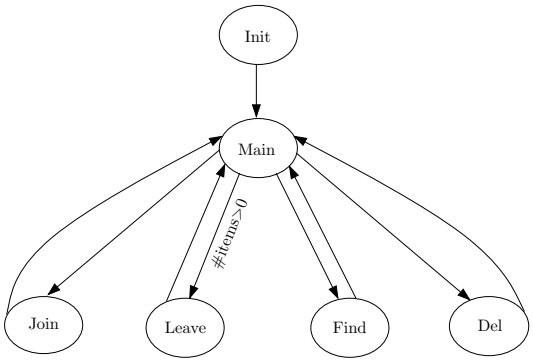

Figure 6.5: Testchart for modified *TimedQueue*

Furthermore, Figure 6.2 and 6.5 show the original version and modified version of the testchart for the *Timed Message Queue System*, respectively. The original *Timed Message Queue System* has the following operations: *Join, Leave, RecLost* and *Del*. The modified version has 3 of the 4 original operations; *RecLost* has been deleted. A new operation *Find* is added. Given an i of type *MSG*, operation *Find* will find out whether it is in the queue. In the modified version of the *Timed Message Queue System*, the MAIN process is defined as follows:

$$\text{MAIN} \mathrel{\widehat=} \mu\, Q \bullet (Join \,\Box\, Leave \,\Box\, Find) \triangleright \{T_o\}\ Del;\ Q$$

Correspondingly, in the testchart for the modified *Timed Message Queue System*, the vertex representing operation *RecLost* doesn't exist while a vertex representing operation *Find* is added. There are three new directed edges: (*Main, Find*), (*Find, Main*) and (*Main, Del*). The following four sets can summarize the differences between the two testcharts:

- **operations-added** = {Find}

- **operations-deleted** = {RecLost}
- **interactions-added** = {(Main, Find), (Find, Main), (Main, Del)}
- **interactions-deleted** = {(Main, RecLost), (RecLost, Del)}

⟨*Init*; *Join*; *Leave*; *RecLost*; *Del*⟩, ⟨*Init*; *RecLost*; *Del*⟩, ⟨*Init*; *Join*; *Leave*⟩ are three test sequences from the test suite that is used to test original *Timed Message Queue System*. Since operation *RecLost* has been deleted, operation sequence ⟨*Init*; *Join*; *Leave*; *RecLost*; *Del*⟩ becomes invalid for the modified *Timed Message Queue System* and cannot be used for regression testing. Because (*Main*, *RecLost*) has been deleted, operation sequence ⟨*Init*; *RecLost*; *Del*⟩ becomes obsolete and cannot be used for regression testing. ⟨*Init*; *Join*; *Leave*⟩ is selected for regression testing because it is valid for the modified *Timed Message Queue System* and one of the operations it includes, *Join*, has been modified. Moreover, new operation *Find* is added; to satisfy the *interaction coverage* criterion, new test sequences such as ⟨*Init*; *Join*; *Find*⟩ need to be generated for regression testing.

6.5 TCOZ-based Regression Test Suite Construction System

In this section, a TCOZ-based regression test suite construction system, which is named TcozRts, is presented. It is built based on the technique that has been introduced in the previous sections.

TcozRts takes the original and modified versions of the specification and the original

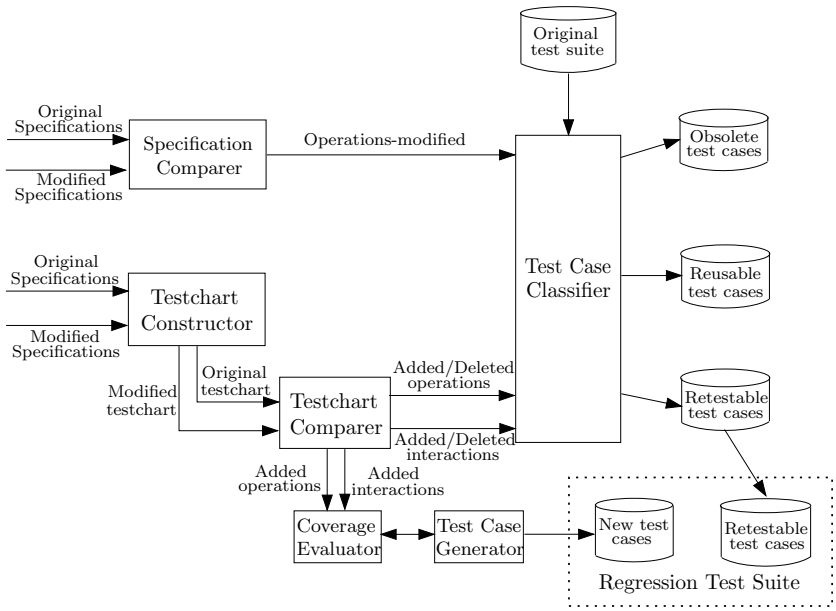

Figure 6.6: TCOZ-based regression test suite construction system: TcozRts

test suite as input, and outputs regression test suite which consists of retestable original test cases and newly generated test cases.

Figure 6.6 shows the architecture of the TCOZ-based regression test suite construction system, TcozRts, and the interactions between the components of the system. *Specification comparer* takes original and modified versions of the specification and identifies the modified operations by automatically parsing the two versions of the TCOZ specification. The information about the modified operations is sent to the *test case classifier*. Meanwhile, the *testchart constructor* constructs testcharts based on the two versions of the specification. The *testchart comparer* compares the two versions of the testchart to identify the added/deleted operations and the added/deleted interactions and sends the information about the modifications to *test case classifier*. With the information from *specification comparer* and *testchart*

comparer, the *test case classifier* categorizes the original test cases as being obsolete, reusable or retestable. The retestable test cases will be used for regression testing. Moreover, it is likely that new functionalities are added to the system; new test cases that can be used to test those parts of the system should be developed for regression testing. In TcozRts, the *test generator* module is design to generate new test cases for the testing of those new operations and interactions, in collaboration with *coverage evaluator*, based on the information about new operations and interactions obtained from *testchart comparer*.

6.6 Conclusion

In this chapter, a formal specification-based regression test suite construction technique for classes specified in TCOZ notation is presented. The proposed technique mainly addresses the *regression test selection problem* and *test suite augmentation problem* involved in a typical selective retest technique. In order to effectively select test cases for regression testing, a TCOZ specification-based regression test selection algorithm is presented. It selects test cases related to the parts of specification that have changed, from the original test suite, based on the changes that have been made to the original version of the formal specification. To solve test suite augmentation problem, our technique compares the two versions of the class' specification to identify the differences and uses those differences to guide the generation of new test cases for regression testing. Furthermore, a TCOZ specification-based regression test suite construction system, TcozRts, is also presented. The technique

presented in this chapter is strictly specification-based and it does not require any complex static or dynamic code analysis. Therefore, it is independent of the programming language which is used to implement the specification, and it can be used as a complement to those code-based regression testing technique to achieve more effective and more comprehensive regression testing in the development and maintenance of software systems.

Chapter 7

Formal Specification Notation for AOSD

With the expectation that the existing formal methods could be extended and applied to aspect-oriented programs, this chapter proposes a formal specification notation, AspecTCOZ, for aspect-oriented software development.

7.1 Introduction

The intent of aspect-orientation is to allow developers to encapsulate and modularize system behaviors that would otherwise tangle and scatter across other concerns; and it is aimed at breaking the hegemony of the dominant decomposition [33, 100]. The research in aspect-oriented software development (AOSD) areas has extended from the design and implementation of aspect-oriented programming languages [52, 57, 99, 18, 108, 107, 96] to the development of techniques that are required in the early stages of software development such as requirement engineering and software design [10, 86, 94]. However, comprehensive supports for aspect-oriented software development are still far from sufficient. Existing formal methods cannot be applied to AOSD directly because new concepts and constructs, such as pointcut, advice, inter-type declaration and aspect, are introduced to aspect-oriented program(AOP).

With the expectation that the existing formal methods could be extended and applied to aspect-oriented programs, this chapter proposes a formal specification notation, AspecTCOZ, for aspect-oriented software development. AspecTCOZ is an aspect-orientation extension to the integrated formal notation TCOZ (Timed Communicating Object-Z) [66, 70], complying with the semantics of AspectJ.

As introduced in Chapter 2, TCOZ is built on the strength of Object-Z [30, 92] in modeling complex data and state with the strength of TCSP [89, 90] in modeling process control and real-time interactions. The class schema in TCOZ notation

is an eligible candidate for specifying aspect formally because *aspect* is the unit of modularity, encapsulation, and abstraction in AOP, just in the same way as *class* in OOP. Meanwhile, the strength of TCSP in modeling process control and real-time interactions, which is preserved in TCOZ, provides a great mechanism for specifying the temporal order between pointcut and advice. Therefore, we try to extend TCOZ with the mechanisms for formally specifying the constructs of join point, pointcut, advice, and inter-type introduction. AspecTCOZ, as the result of the aspect-orientated extension of TCOZ, provides a starting point for future research work on the development of formal methods for aspect-oriented software development.

Moreover, in AOP, when multiple aspects are superimposed on the same join point, undesire or incorrect behavior may emerge due to unexpected conflicts between aspects. The conflicts resulting from the introduction of aspects are usually implicit and difficult to capture specially when the program control flow is influenced by those conflicts. The development of effective mechanisms for detecting those conflicts between aspects is critical to the maturity of AOP. Furthermore, early detection of those conflicts will make it possible to reduce the development cost while promising a high quality software system. Therefore, this chapter proposes an approach for the detection of conflicts between aspects, based on the formal specification of the system which is written in AspecTCOZ notation, so that the aspect conflicts can be detected as early as in the phase of system design.

The rest of this chapter is organized as follows. Section 7.2 introduces a simple

telephone system which will be taken as an example in this chapter. Section 7.3 proposes AspecTCOZ notation, which is an extension to TCOZ notation with the mechanisms for formally specifying the constructs of join point, pointcut, advice, and inter-type introduction. Section 7.4 proposes an approach for the early detection of conflicts between aspects, based on the formal specification of the system which is written in AspecTCOZ notation. Finally, Section 7.5 concludes this chapter.

7.2 Overview of a Simple Telephone System

A simple simulation of a telephony system, which is part of the AspectJ distribution, will be taken as an example in the following sections. In the simple telephone system, customers make, accept, and hang-up both local and long distance calls.

The application architecture consists of:

- The basic objects provide basic functionality to simulate *customers*, *calls* and *connections* (regular calls have one connection, conference calls have more than one).
- The *timing* feature is concerned with timing the connections and keeping the total connection time per customer. Aspect is used to add a timer to each connection and to manage the total time per customer.
- The *billing* feature is concerned with charging customers for the calls they make. Aspect is used to calculate a charge per connection and, upon ter-

$STATE ::= pending \mid complete \mid dropped$

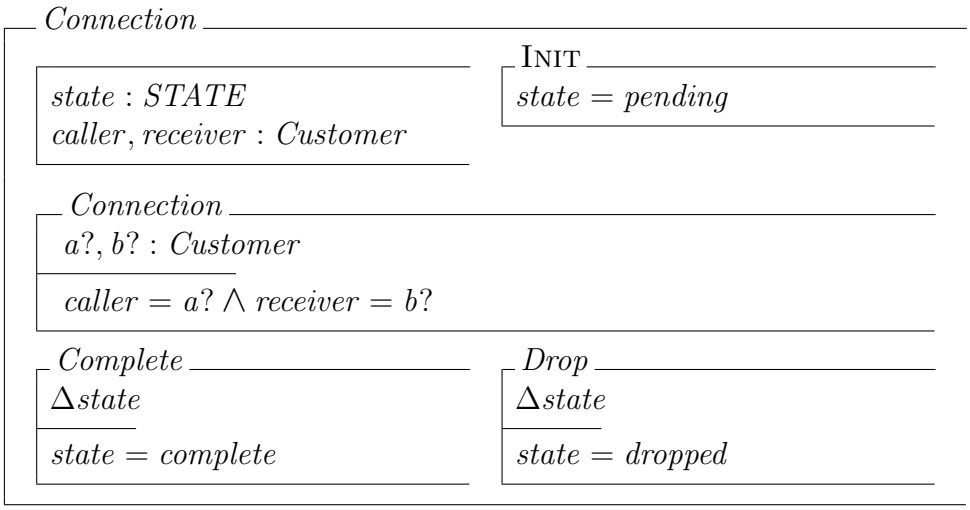

Figure 7.1: The *Connection* class

mination of a connection, to add the charge to the appropriate customer's bill.

The definition of the *Connection* class in TCOZ notation is shows in Figure 7.1. It describes that there should be a caller and a receiver in order to establish a connection, and that the connection could be *pending* when the receiver has not picked up the phone, *complete* when the caller and receiver finish calling, or *dropped* when either caller or receiver drop the phone arbitrarily.

The definition of the *Customer* class in TCOZ notation is shown in Figure 7.2. The specification describes that a customer has a name, an area code and a record of the calls; and he or she can make a call, accept a call and modify the record of the calls simultaneously. Note that the operations *MakeCall, AcceptCall* and *Main* are defined in terms of CSP processes while the operations *AddCall* and *RemoveCall*

$[NAME]$

Customer

$name : NAME$
$areacode : \mathbb{N}$
$calls : \mathbb{P}\, Call$
$T_0 : \mathbb{N}$

INIT
$calls = \varnothing$

AddCall
$\Delta(calls)$
$c? : Call$

$c? \notin calls \land c?.Init$
$calls' = calls \cup \{c?\}$

RemoveCall
$\Delta calls$
$c? : Call$

$c? \in calls$
$calls' = calls \setminus \{d?\}$

$MakeCall \mathrel{\widehat=} AddCall;\ [c : calls] \bullet c.PickUp;\ ((c.Connected;\ c.Hangup) \Box c.NotConnected \triangleright \{T_0\}\ (c.HangUp;\ RemoveCall))$
$AcceptCall \mathrel{\widehat=} AddCall;\ [c : calls] \bullet c.PickUp;\ c.Connected;\ c.HangUp$
$\textsc{Main} \mathrel{\widehat=} \mu\, Q \bullet (MakeCall \Box AcceptCall);\ Q$

Figure 7.2: The *Customer* class

are defined in form of operation schemas. Besides, the state variable *call* is a set of instances of class *Call* which is the abstract description of connections between a caller and receiver who are customers.

7.3 AspecTCOZ - an Extension of TCOZ

Aspect is the central unit of modularity, encapsulation, and abstraction in AOP, in the same way that *class* is in OOP. It is defined very much like a class, and can contain methods, fields, nested class members, and initializers, just like a normal OOP class. The data members and methods inside aspects function in the same way they do in classes. The crosscutting concern could manage its state using the

data members, whereas the methods could implement behavior that supports the crosscutting concern's implementation, or they could simply be utility methods. Moreover, just like class inheritance in OOP, there is a mechanism for aspects inheritance with which aspects can not only extend other aspects, but also extend classes and implement interfaces in AOP. Thereby, the class schema in TCOZ notation, as presented in Section 2.1.2, might be an eligible candidate for specifying *aspect* formally.

However, as the basic units for implementing aspect-oriented crosscutting concerns, aspects must contain the constructs that express the weaving rules for both dynamic and static crosscutting, such as pointcuts, advice, and inter-type declarations and so on. There are no such mechanisms in TCOZ that can specify those aspect-orientation constructs properly. Therefore, we extend TCOZ notation with mechanisms for specifying the constructs introduced by aspect-oriented programming, to provide a starting point for future research work on the development of formal methods for aspect-oriented software development.

7.3.1 Join Point

Join points are events in the control flow of a program. They are identifiable points in the execution of a program. In AOP, everything revolves around join points, since they are the places where the crosscutting actions are woven in. A join point model determines which events will be exposed as join points, and which will not. The join points defined by AspectJ include: method and constructor calls

Table 7.1: Formal notations for join point model of AspectJ

Join point model	Formal notation	Example
method call constructor call	ζ(list of *operation/constructor signatures*)	ζ(AddCall)
method execution constructor execution handler execution advice execution object initialization class initialization	ξ(list of *operation/constructor/handler/ advice/object initialization signature*)	ξ(RemoveCall)
field read access	Ⓡ(list of *class member names*)	Ⓡ(name)
field write access	Ⓢ(list of *class member names*)	Ⓢ(areacode)

or executions, field accesses, object and class initializations, handler and advice executions, and so on [57]. In AspecTCOZ, we introduce formal notation, which is as show in Table 7.1, for the join point model of AspectJ.

The examples in Table 7.1, namely ζ(AddCall), ξ(RemoveCall), Ⓡ(name), Ⓢ(areacode), describe respectively the join point of calling the method described by the operation *AddCall*, executing method described by the operation *RemoveCall*, getting the value of the field *name*, and setting the value of the field *areacode*.

7.3.2 Pointcut

Join point selections are of great importance in aspect-oriented software development. They designate all those relevant points in a program at which aspectual adaptations need to take place. Different aspect-oriented systems come up with most various language constructs to specify such selections. For example, in Hyper/J [99], *match patterns* designate method specifications based on their names. In Sally [80], *logic queries* select classes based on particular attributes or methods

Table 7.2: Formal notations for pointcut designators of AspectJ

Pointcut designator	Formal notation	Example
control-flow based pointcuts	▼(*executing/calling pointcuts*)	▼(ζ(AddCall))
	▽(*executing/calling pointcut*)	▽(ξ(RemoveCall))
lexical-structure based pointcuts	◯(name of class)	◯(Customer)
	◎(name of operation)	◎(MakeCall)
execution object poincuts	⊕(name of class/object)	⊕(Customer)
	⊙(name of class/object)	⊙(Customer)
argument pointcuts	$\mathcal{A}$(type/name of parameters)	$\mathcal{A}$(Call)
conditional check pointcuts	predicates from TCOZ	$c? \notin calls$

that they contain. In JAsCo [29], *applicability conditions* select objects depending on the state they are in.

In AspectJ, the language construct for join point selection is *pointcut*. A pointcut is a program construct that captures a set of join points by matching certain characteristics. It acts as a filter, matching join points that meet its specification, and blocking all others.

In addition to the join points which have been presented in Section 7.3.1, the pointcut designators in AspectJ can also capture join points based on matching the circumstances under which they occur, such as control flow, lexical scope, and conditional checks. Table 7.2 presents the proposed formal notation for pointcut designators.

Among the examples, ▼(ζ(AddCall)) describes the pointcut that selects all the join points in the control flow of the method which corresponds to the operation *AddCall* as defined in class *Customer*, including the call to the method itself. ◯(Customer) describes the pointcut that selects all the join point inside the *Customer* class's

lexical scope. $\oplus$(Customer) describes the pointcut the capture all the join points like methods calls and field assignments where the current execution object is *Customer*, or its subclass. $\mathcal{A}(Call)$ describes all the join points in all operations where the argument is of type *Call*. The predicates of TCOZ act as the same as what conditional check pointcuts do in AspectJ, therefore, we do not introduce new formal mechanism for it. $c? \notin calls$ describes the pointcut which captures all the join points where a new call is started.

Complex matching rules can be formed by combining simple pointcuts. Like in AspectJ, AspecTCOZ provides a unary negation operator $\neg$ and two binary operators $\wedge$ and $\vee$ to build powerful pointcuts from the simple building blocks of existing and primitive pointcuts.

- unary negation operator $\neg$ allows the matching of all join points except those specified by the pointcuts.

  For example, $\neg(\bigcirc(\text{Customer}))$ excludes all the join points inside the *Customer* class's lexical scope.

- $\wedge$ and $\vee$ are provided to combine pointcuts. Combining two pointcuts with the $\vee$ operator causes the selection of join points that match either of the pointcuts, whereas combining them with $\wedge$ operator causes the selection of join points matching both the pointcuts.

  For example, $\odot(Customer) \wedge \zeta(Customer.AddCall)$ describes the pointcut that captures the join points where the object on which the method called is an instance of *Customer*, meanwhile, the method corresponding to the

operation *AddCall* is called.

To provide a sound formal notation, the following two important characteristics of the construct of pointcut should be noted.

- In AspectJ, pointcuts can be named or anonymous. In AspecTCOZ, we demand that every pointcut has a name. Naming pointcuts provides the software designers the ability to abstract, encapsulate and maximize the reusability of join point selections in different application contexts. Also, it improves the clarity of the specification documents.

- A pointcut can also collect context at those join points it selects through some parameters. In AOP, an advice declaration may contain parameters whose values can be referenced in the body of the advice. However, because advices cannot be called by name, or by any other means, parameter values cannot be explicitly passed by the caller like in method calls. Therefore, the parameter provided by the pointcut is an essential way for the advice to get information about context.

In AspecTCOZ, the general form of a pointcut declaration is as follows:

$$PointcutName\ [(ParameterTypeList)] \triangleq$$
$$PrimitivePointcut\ \{\neg \mid \wedge \mid \vee\ (PrimitivePointcut | PointcutName)\}$$

Literally, the pointcut can have zero or a list of parameters; and it can be a primitive pointcut or the combination of existing pointcuts and/or primitive poincuts with operators (i.e. $\neg, \wedge, \vee$).

For example, the following AspecTCOZ statement describes a pointcut that is named as *endTiming* and has a parameter which is of type *Connection*.

$$endTiming(Connection) \triangleq \odot(Connection) \wedge \xi(Connection.Drop)$$

This pointcut captures the join points where the object, on which the method is called, is an instance of *Connection*; meanwhile, the method corresponding to the operation *Drop* is executed.

7.3.3 Advice

Advice is a method-like construct that provides a way to express what to do at the join points that are captured by a pointcut, and it is the action and decision part of the crosscutting concern. The operation schema in TCOZ work greatly in describing "what to do". However, it is not capable enough of formally specifying advice because each piece of advice must be associated with a pointcut in AOP.

The implicit invocation of advice can happen *before* the join points matched by its pointcut, *after* the join points matched by its pointcut, or *around* the join points. *Before* and *after advice* are the simple kinds: whereas they can read contextual information at a join point (such as arguments and return values), they cannot change it. *Around* advice is the most powerful form of advice; it can not only read contextual information but also change it and can even decide whether the original join point should be executed at all.

Now, we exploit the strength of TCOZ notation to formally specify *advice*. Firstly,

pointcut captures a set of join points by matching certain characteristics while join points captured by the pointcuts are essentially events in the execution of a program. Therefore, we can define a process *PCprocess* for every pointcut *PointCut* as follows:

$$PCprocess = e : PointCut \rightarrow SKIP$$

Secondly, as introduced in Section 2.1.2, in TCOZ notation, operation schemas (both syntactically and semantically) is identified with (terminating) CSP processes that perform only state update events; and operation schema expressions may appear wherever processes may appear in CSP. Meanwhile, the strength of TCSP in modeling process control and real-time interactions, which is preserved in TCOZ, provides a great mechanism for specifying the temporal order between pointcut and advice. Therefore, assuming *OP* is the operation schema describing "what to do" with the advice and *PCprocess* is the process corresponding to the relevant pointcut, we can specify the *before advice* and *after advice* as the sequential composition of two processes, namely *PCprocess* and *OP*, as follows:

$$\textbf{before advice} \Rightarrow OP;\ PCprocess$$

$$\textbf{after advice} \Rightarrow PCprocess;\ OP$$

As an example, the specification in Figure 7.3 describes an *after advice* named *CalculateCost*. Following the join point captured by the pointcut $endTiming(Connection)$, the system is designed to do what is described by the operation schema *CalCost*. This advice requires that, after the termination of a connection, the charge for the

$$
\begin{array}{|l}
\hline
\textit{CalCost} \\
\textit{conn?} : \textit{Connection} \\
\textit{time}, \textit{rate}, \textit{cost} : \mathbb{N} \\
\hline
\textit{cost} = \textit{rate} * \textit{time} \\
\textit{conn.payer.cost}' = \textit{conn.payer.cost} + \textit{cost} \\
\hline
\end{array}
$$

$$\textit{CalculateCost} = e : \textit{endTiming}(\textit{Connection}) \rightarrow \textit{SKIP};\ \textit{CalCost}$$

Figure 7.3: An *after* advice

connection is calculated and added the charge to the appropriate customer's bill.

While specifying advice with AspecTCOZ notation, the following two points should be noted:

- First, in AspecTCOZ notation, it is compulsory that every advice has a name associated with it in order that we can achieve the benefit of abstraction and encapsulation.
- Second, each parameter in the operation schema must appear somewhere in the definition of the pointcut which is associated with the advice since advice gets contextual information at a join point through pointcut.

As illustrated by the example in Figure 7.3, this *after advice* does have a name which is *CalculateCost*. Meanwhile, the input parameter of the operation schema *CalCost*, i.e. *Connection*, is indeed included in the parameter list of the pointcut *endTiming*(*Connection*).

7.3.4 Inter-type Declaration

Whereas advice is a declaration that an aspect will execute certain behavior in the program control flow at designated join points, inter-type declarations are statements that an aspect takes complete responsibility for certain capabilities on behalf of the "targets" of the inter-type declarations.

The most basic forms of inter-type declarations are for methods, fields, and constructor. An inter-type declaration inside an aspect looks just like the definition of a normal method, field in constructor in the aspect. The mechanisms provided by AspecTCOZ for specifying inter-type declaration of fields, methods and constructors are similar to those mechanisms for normal declarations, but with the exception that the targets of the declarations are attached with sign '$\propto$'. There are two purposes to do so:

1. to distinguish the inter-type declarations from the normal ones;

2. to show clearly what the target modules of the inter-type declarations are.

The the general form of inter-type declaration of state variables in AspecTCOZ is as follows:

$$\propto$$
$$\{NameOfVariable \propto TargetClassName : Type\}_1$$
$$[Predicates \quad on \quad Variables]$$

As an example, the following AspecTCOZ specification describes the inter-type declaration that class *Connection* has an inter-type field, *payer*, to indicate who

initiates the call and therefore is responsible to pay for it, and that class *Customer* has an inter-type field, *totalConnectioinTime*, to store the accumulated connection time for every customer and an inter-type field, *totalCharge*, to store the accumulated charge that the customer should pay. The predicate at the bottom of the schema indicates that the value of the field *totalCharge* is never less than 0.

$$
\begin{array}{|l}
\hline
\propto \\
payer \propto Connection : Customer \\
totalConnectionTime \propto Customer : \mathbb{N} \\
totalCharge \propto Customer : \mathbb{R} \\
\hline
totalCharge \propto Customer \geqslant 0 \\
\hline
\end{array}
$$

The the general form of inter-type declaration of operation in AspecTCOZ is as follows:

$$
\begin{array}{|l}
\hline
OpName \propto TargetClassName \\
[ListOfToBeChanged] \\
[NameOfVariable \propto TargetClassName : Type] \\
\hline
[Predicates \ \ on \ \ Variables] \\
\hline
\end{array}
$$

As an example, the following AspecTCOZ specification declares an inter-type method which targets *Customer* class and updates the field *totalCharge* with the new charge included and keeps all the other fields of the *Customer* class unchanged.

$$
\begin{array}{|l}
\hline
addCharge \propto Customer \\
\Delta(totalCharge) \\
charge? : \mathbb{R} \\
\hline
totalCharge' = totalCharge + charge? \\
\hline
\end{array}
$$

7.3.5 Aspect

The aspect is the central unit of AOP, in the same way that a class is the central unit in OOP. It contains the code that expresses the weaving rules for both dynamic and static crosscutting. Pointcuts, advice, introductions, and declarations are combined in an aspect. Besides, aspects can contain data, methods, and nested class members, just like a normal class. Moreover, just like class inheritance in OOP, there is a mechanism for aspects inheritance with which aspects can not only extend other aspects, but also extend classes and implement interfaces in AOP.

Having proposed the extension to TCOZ notation for formally specifying join point, pointcut, advice, and inter-type introduction, now we can formally specifying an aspect with AspecTCOZ notation. The general form of an aspect schema is as follows.

AspectName

[*Inheritance*]
[*LocalDefinition*]
[*StateSchema*]
[*InitialSchema*]
[*OperationSchema*]
[*OperationExpressionDefinition*]
[*PointcutDefinition*]
[*AdviceDefinition*]
[*IntertypeStateVariableSchema*]
[*IntertypeStateVariableInitialSchema*]
[*IntertypeOperationSchema*]

As an example, the specification shown in Figure 7.4 describes an aspect named

TimingBilling. This aspect will perform the following functions: keeping the total connection time per customer, and charging customers for the calls they make.

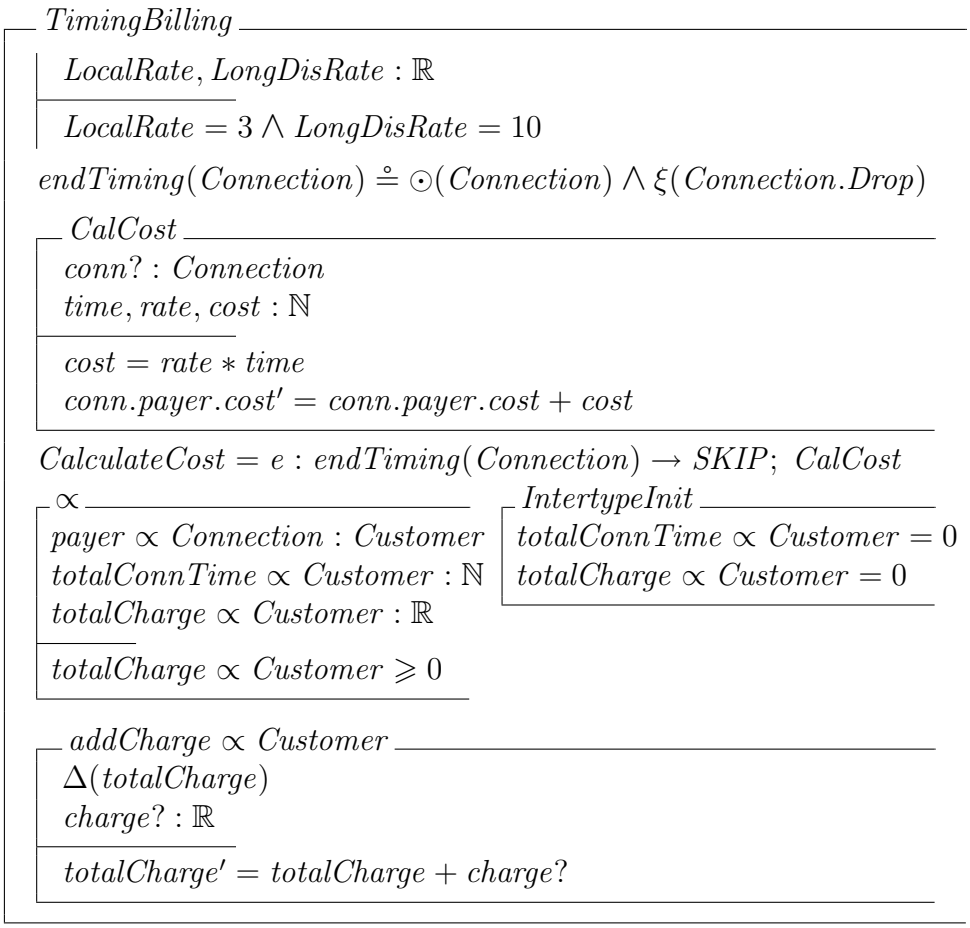

Figure 7.4: *TimingBilling* aspect in AspecTCOZ notation

7.4 Formal Specification-based Aspect Conflict Detection

Aspect-oriented software development supports multi-dimensional separation of concerns throughout the software development cycle. It is a promising methodology aiming to enhance the productivity, quality and reusability through the en-

capsulation of requirements that cut across core concerns. However, there are also intrinsic and critical issues in aspect-oriented software development. One of them is the aspect conflicts problem.

In AOP, it is allowed that multiple aspects are superimposed on the same join point. When multiple aspects are superimposed on the same join point, the aspects might well interfere with each other in a potentially undesired manner.

This kind of issues are extremely hard to detect, as those aspects are syntactically sound, and will be compiled without any problems. The conflicts exhibit themselves only when the composed application executes. It might be caused by the side effects of behavior of the aspects at the join point, for example, the aspects might change the state of the base program. Also, it might be caused by the requirements enforced by the system, for example, the logging aspect may be applied only in the presence of the encryption aspect because the systems require all logged data to be encrypted. For the latter case, the conflicts cannot be detected without extra information about the specific application requirements.

The detection of aspect conflicts has been considered as an important issue to aspect-oriented software development and has received attention from researchers. Durr *et al.* [31] proposed a detecting approach that defines the semantics of advice in terms of operations on a resource model. After all advice at a shared join point has been analyzed, the conflicts will be detected based on conflict patterns over the combinations of operations on these resources. Tessier *et al.* [101] proposed a formal way to detect semantic conflicts between aspects based on extended UML

class diagram model. In their approach, the relationship between aspects and classes is translated into formal rules through model analysis; then a rule analyzer evaluate those rules to detect eventual conflicts between aspects.

In this section, we propose an approach for detecting aspect conflicts as early as in the phase of system design. Our approach is based on the formal specification of the system, which is written in AspecTCOZ notation.

In AspecTCOZ, the aspect is defined by the association of operation schema and pointcut. The operation schema describes clearly "what to do" at the join point. In the operation schema, the input and output parameters are clearly laid out, and the variables/objects that will be changed by the operation are also explicitly laid out in the *ListOfToBeChanged*. Thus, based on the formal specification, we can figure out whether there might be any data-dependent conflicts between aspects. Assume that a system has been specified in AspecTCOZ notation, and that there are some join points which are superimposed by a few aspects. For each join point, if it is superimposed then for each pair of the aspects superimposing on it (A_1, A_2), for each advice a_1 which is an advice included in aspect A_1, we check whether there exists an advice a_2 in aspect A_2 such that advice a_2 is the same kind of advice as a_1 and there are variables which are included both in the input variable list of the operation schema associated with a_1 and in the output variable list or the *ListOfToBeChanged* of the operation schema associated with a_2. If yes, there will be conflicts between the two aspects A_1 and A_2. We also check whether there are common elements between the *ListOfToBeChanged* of the operation schema

associated with a_1 and the *ListOfToBeChanged* of the operation schema associated with a_2. If yes, we declare that there will be conflicts between A_1 and A_2.

7.5 Conclusion

In this chapter, we propose AspecTCOZ, which is an aspect-orientated extension to the integrated formal notation TCOZ. Resulting from extending TCOZ notation with the mechanisms for formally specifying the constructs of join point, point-cut, advice, and inter-type introduction, AspecTCOZ provides a starting point for future research work on the development of formal methods for aspect-oriented software development. Furthermore, we propose an approach for handling aspect conflicts problem. Based on the formal specification of the system, which is written in AspecTCOZ notation, our approach can detect data-dependent conflicts between aspects as early as in the design and modeling phase of system development. It helps in reducing the development cost while promising a high quality software system.

Chapter 8

Conclusion

This chapter serves two purposes. Firstly, it summarizes the main contributions of the whole thesis. Secondly, it provides a discussion on some possible directions for future research.

8.1 Main Contributions of the Thesis

The main contributions of this thesis are summarized as follows:

- **A formal specification-based software monitoring technique**

  Based on formal specification animation and program debugging, our formal specification-based monitoring technique dynamically gathers required information, interprets the gathered information and responds appropriately in a timely manner as the target system is running. It can not only dynamically and continuously monitor the behaviors observed in the target system, but also explicitly recognize undesirable behaviors in the target system with respect to given formal requirement specifications. Our formal specification-based monitoring technique can contribute to increasing the dependability, correctness, robustness and security of the target system. It is a good candidate to be used as a complementary technique to formal verification. Moreover, it can also support software testing in automatic test execution and in checking whether actual output of the program under test is equivalent to the expected output.

- **An AOP-aided software evolution approach**

  Based on the *weaving* mechanism and *pointcuts* construct of AOP (aspect-oriented programming), the proposed AOP-aided evolution approach is capable of handling the evolution of core classes in object-oriented programs, when the formal specification of the system has changed.

- **A formal specification-based regression test suite construction technique**

  The proposed technique mainly addresses the *regression test selection problem* and *test suite augmentation problem* that are involved in a typical selective retest technique. It selects test cases that will be reused for regression testing, from the original test suite, according to the changes that have been made to the original formal specification of the system. To solve test suite augmentation problem, it guides the generation of new test cases for regression testing with the differences between the original and new versions of the system's formal specification. The proposed technique is strictly specification-based and it does not require any complex static or dynamic code analysis. It can be used as a complement to those code-based regression testing technique to achieve more effective and more comprehensive regression testing in the development and maintenance of software systems.

- **A formal specification notation for AOSD - AspecTCOZ**

  Resulting from the extension of TCOZ with the mechanisms for formally specifying the constructs of join point, pointcut, advice, and inter-type introduction, AspecTCOZ provides a starting point for future research work on the development of formal methods for aspect-oriented software development.

- **A formal specification-based aspect conflicts detection approach**

  Based on AspecTCOZ notation, we propose a formal specification-based aspect conflicts detection approach. Our approach can detect the conflicts

between aspects as early as in the design and modeling phase of system development. It helps in reducing the development cost while promising a high quality software system.

8.2 Future Work Directions

Based on the work presented in this thesis, there are a few possible directions for future research, which may further exploit the potentials of formal specification in benefiting the development and maintenance of various software systems. In this section, some of those possible directions are discussed briefly.

8.2.1 Further Development of Monitoring Technique

At present, our formal specification-based monitoring technique works at intra-class level, it can detect the incorrect implementation of methods in the class. To improve the monitoring technique so that it can work at inter-class level and detect errors caused by the improper invocations of methods between classes is a part of future work.

The prototype monitoring system that we have developed makes judgement of the conformance of implementation with formal specification based on the current value of class' data members. It can handle the situation where the return value is of simple types. It needs to be improved to be able to deal with the situation where the return value is `object`.

We have extended the monitoring system to work with aspect-oriented programs.

However, currently, it can only detect the errors in the programs resulting from weaving the aspects with base programs but can not figure out the aspects or advices that are the cause of the error. To further develop the monitoring technique so that it can accurately figure out the incorrect implementation of which aspect or advice leads to the error will be a great contribution to aspect-oriented software development.

Furthermore, monitoring distributed and parallel system during execution can provide information that can be used to reconfigure the system, provide visualization of behavior, or steer its outcome [91]. Therefore, we also intend to extend our monitoring technique so that it can handle distributed and parallel systems.

8.2.2 Formal Methods for AOSD

In the formal specification notation AspecTCOZ, which is proposed in Chapter 7, the strength of TCSP in modeling process control and real-time interactions, which is preserved in TCOZ, is used for specifying *before* and *after* advice. However, it is not capable of formally specifying *around* advice by far. More delicate mechanism is required to be introduced for specifying *around* advice in the future.

Meanwhile, AspecTCOZ complies with AspectJ which shares with other aspect-oriented languages a common core principle: an aspect contains definition of behaviors (*advice*) and specifications of where the behaviors should be executed (*pointcuts*); pointcuts are quantified statements over a program, selecting a set of well-defined points (*joinpoints*) during the execution of a program. Therefore, one

possible direction to improve AspecTCOZ is to further investigate the underlying fundamentals of aspect-oriented programming and come up with more abstract and generic formal constructs for aspect oriented software design. Meanwhile, the clear and rigorous definition of the semantics of AspecTCOZ needs lots of effort in future research.

With the formal specification available, the development tool supports for aspect-oriented software verification and validation will be of great value to aspect oriented software development. The research in this direction deserves the attentions and efforts of researchers. The development of an animator for the formal specifications written in AspecTCOZ would be part of our future work. With an AspecTCOZ specification animator, the formal specification-based monitoring technique that we have proposed would work more efficient for the validation of aspect-oriented programs.

As introduced in Chapter 7, when multiple aspects are superimposed on the same join point, the aspects might well interfere with each other in a potentially undesired manner. We have proposed a conflicts detection approach based on a system's formal specification, in Chapter 7. However, the proposed approach can only detect data-dependent conflicts. To detect more implicit conflicts, a detection technique which is based on control dependence analysis is required. Detecting aspect conflicts as early as in the phase of software design and modeling will prevent them from propagating through latter phases of software development, and reduce the development cost remarkably. Therefore, it is worthwhile to develop

a powerful formal specification-based detection technique that can deal with both data-dependent and control-dependent conflicts.

Bibliography

[1] AspectJ. http://www.eclipse.org/aspectj.

[2] JCrasher. http://www.cc.gatech.edu/jcrasher/.

[3] jdb - The Java Debugger. http://java.sun.com/.

[4] Parasoft Jtest. http://www.parasoft.com/jtest.

[5] J. R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[6] R. Achuthan, V. S. Alagar, and T. Radhakrishnan. An object-oriented modeling of real-time robotic assembly system. In *Proceedings of the 1st IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '95)*, pages 310–313, 1995.

[7] V. S. Alagar and G. Ramanathan. Functional specification and proof of correctness for time dependent behaviour of reactive systems. *Formal Aspect of Computing*, 3(3):253–283, 1991.

[8] A. A. Andrews, R. B. France, S. Ghosh, and G. Craig. Test adequacy criteria for UML design models. *Software Testing, Verification & Reliabbility.*, 13(2):95–127, 2003.

[9] T. Ball. On the limit of control flow analysis for regression test selection. In *ACM International Sympodium on Software Testing and Analysis*, pages 134–142, 1998.

[10] E. Baniassad and S. Clarke. Theme: An approach for aspect-oriented analysis and design. In *Proceedings of the 26th International Conference on Software Engineering*, pages 158–167, 2004.

[11] D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim. Jass - Java with assertions. In *Proceedings of First Workshop on Runtime Verification, RV'01*, 2001.

[12] B. Beizer. *Software Testing Techniques*. Von Nostrand Reinhold, 1990.

[13] D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[14] Dines Bjørner. *The SE Book: Principles and Techniques of Software Engineering*. Springer-Verlag, 2004.

[15] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[16] R. M. Burstall and J. A. Goguen. The semantics of CLEAR, a specification language. *Lecture Notes in Computer Science*, 86:293–329, 1980.

[17] D. Carrington, D. Duke, R. Duke, P. King, G. Rose, and G. Smith. Object-Z: An object-oriented extension to Z. In S. Vuong, editor, *Formal Description Techniques, II (FORTE'89)*, pages 281–296, 1990.

[18] K. Chen, S. C. Weng, M. Wang, S. C. Khoo, and C. H. Chen. A compilation model for aspect-oriented polymorphically typed functional languages. In *Proceedings of the 14th International Static Analysis Symposium*, 2007. To appear.

[19] Y. F. Chen, D. S. Rosenblum, and K. P. Vo. TestTube: A system for selective regression testing. In *Proceedings of the 16th International Conference on Software Engineering*, pages 211–222, May 1994.

[20] S. Clarke. *Composition of Object-oriented Software Design Models*. PhD thesis, Dublin City University, 2001.

[21] S. Clarke and R. J. Walker. Composition patterns: An approach to designing reusable aspects. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 5–14, 2001.

[22] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Using Maude. In *Proceedings of the Third Internationsl Conference on Fundamental Approaches to Software Engineering*, pages 371–374, 2000.

[23] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 65–89, 1996.

[24] C. Csallner and Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software: Practice & Experience*, 34(11):1025–1050, 2004.

[25] S. A. Curtis, J. Mica, J. Nuth, G. Marr, M. L. Rilee, and M. K. Bhat. ANTS (Autonomous Nano-Technology Swarm): An artificial intelligence approach to asteroid belt resource exploration. In *Proceedings of International Astronautical Federation, 51st Congress*, 2000.

[26] S. A. Curtis, W. F. Truszkowski, M. L. Rilee, and P. E. Clark. ANTS for human exploration and development of space. In *Proceedings of IEEE Aerospace Conference*, 2003.

[27] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30(12):859–872, 2004.

[28] J. S. Dong, J. Colton, and L. Zucconi. A formal object approach to real-time specification. In *Proceedings of the 3rd Asia-Pacific Software Engineering Conference (APSEC'96)*, 1996.

[29] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 141–150, 2004.

[30] R. Duke and G. Rose. *Formal Object Oriented Specification Using Object-Z*. Cornerstones of Computing. Macmillan, March 2000.

[31] P. Durr, L. Bergmans, and M. Aksit. Reasoning about semantic conflicts between aspects. In *Proceedings of the First Aspect, Dependencies, and Interactions Workshop*, pages 10–18, 2006.

[32] H. Ehrig and B. Mahr. Fundamentals of Algebraic Specification 1: Equations and Initial Semantics. *EATCS Monographs on Theoretical Computer Science*, 6, 1985.

[33] R. E. Filman, T. Elrad, S. Clarke, and M. Akist. *Aspect-oriented Software Development*. Addison-Wesley Professional, 2005.

[34] J. Goguen and J. Tardo. An introduction to OBJ: A language for writing and testing software specifications. In N. Gehani and A. McGettrick, editors, *Software Specification Techniques*, pages 391–420, 1985.

[35] T. L. Graves, M. J. Harrold, J. M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208, 2001.

[36] J. V. Guttag and J. J. Horning. *Larch: languages and tools for formal specification*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[37] M. J. Harrold, J. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. Spoon, and A. Gujarathi. Regression test selection for Java software. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications OOPSLA 2001*, pages 312–326, October 2001.

[38] K. Havelund and G. Roşu. Java PathExplorer - a runtime verification tool. In *Proceedings of 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space, ISAIRAS'01*, 2001.

[39] K. Havelund and G. Roşu. Monitoring Java programs with Java PathExplorer. In *Proceedings of First Workshop on Runtime Verification, RV'01*, 2001.

[40] K. Havelund and G. Roşu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.

[41] D. Hazel, P. Strooper, and O. Traynor. Possum: An animator for the SUM specification language. In *APSEC '97: Proceedings of the Fourth Asia-Pacific Software Engineering and International Computer Science Conference*, page 42. IEEE Computer Society, 1997.

[42] D. Hazel, P. Strooper, and O. Traynor. Requirements engineering and verification using specification animation. In *ASE '98: Proceedings of the Thirteenth IEEE Conference on Automated Software Engineering*, page 302. IEEE Computer Society, 1998.

[43] M. A. Hewitt, C. O'Halloran, and C. T. Sennett. Experiences with PiZA, an Animator for Z. In *ZUM '97: Proceedings of the 10th International Conference of Z Users on The Z Formal Specification Notation*, pages 37–51. Springer-Verlag, 1997.

[44] M. G. Hinchey, Y. S. Dai, C. A. Rouff, J. L. Rash, and M. R. Qi. Modeling for NASA autonomous nano-technology swarm missions and model-driven autonomic computing. In *AINA '07: Proceedings of the 21st International Conference on Advanced Networking and Applications*, pages 250–257, 2007.

[45] M. G. Hinchey, C. A. Rouff, J. L. Rash, and W. F. Truszkowski. Requirements of an integrated formal method for intelligent swarms. In *FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 125–133, 2005.

[46] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[47] J. R. Horgan and S. London. Data flow coverage and the c language. In *Proceedings of the symposium on Testing, analysis, and verification*, pages 87–97, 1991.

[48] P. Hsia, X. Li, C-T. Hsu D. Kung, L. Li, Y. Toyoshima, and C. Chen. A technique for the selective revalidation of OO software. *Software Maintenance: Research and Practice*, 9:217–233, 1997.

[49] Jonathan Jacky. *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press, 1997.

[50] I. Jacobson and P. W. Ng. *Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series).* Addison-Wesley Professional, 2004.

[51] C. Kaner, J. Bach, and B. Pettichord. *Lessons Learned in Software Testing.* John Wiley & Sons, Inc., New York, NY, USA, 2001.

[52] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353. Springer-Verlag, 2001.

[53] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[54] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a run-time assurance tool for Java. In *Proceedings of First Workshop on Runtime Verification, RV'01*, 2001.

[55] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Computational analysis of run-time monitoring - fundamentals of Java-MaC. In *Proceedings of Second Workshop on Runtime Verification, RV'02*, 2002.

[56] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O.V. Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *Formal Methods in System Design*, (2):129–155, 2004.

[57] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning Publications Co., Greenwich, CT, USA, 2003.

[58] H. K. N. Leung and L. J. White. Insights into regression testing. In *Proceedings of the International Conference on Software Maintenance*, pages 60–69, 1989.

[59] H. K. N. Leung and L. J. White. A cost model to compare regression test strategies. In *Proceedings of the Conference on Software Maintenance '91*, pages 201–208, October 1991.

[60] H. Liang. Regression testing of classes based on tcoz specification. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2005)*, pages 450–457, 2005.

[61] H. Liang, J. S. Dong, and J. Sun. Evolution and runtime monitoring of software systems. In *Proceedings of the 19th International Conference on Software Engineering and Knowledge Engineering (SEKE 2007)*, pages 343–348, 2007.

[62] H. Liang, J. S. Dong, J. Sun, R. Duke, and R. E. Seviora. Formal specification-based online monitoring. In *Proceedings of the 11th IEEE Inter-*

national Conference on Engineering of Complex Computer Systems (ICECCS 2006), pages 152–160, 2006.

[63] H. Liang and J. Sun. Modular specification of aspect-oriented systems and aspect conflicts detection. In *Proceedings of the 19th International Conference on Software Engineering and Knowledge Engineering (SEKE 2007)*, pages 77–80, 2007.

[64] K. Lieberherr, D. Orleans, and J. Ovlinger. Aspect-oriented programming with adaptive methods. *Commun. ACM*, 44(10):39–41, 2001.

[65] S. Y. Liu, J. Offutt, C. Ho-Stuart, M. Ohba, and Y. Sun. Sofl: A formal engineering methodology for industrial applications. *IEEE Transactions on Software Engineering*, 24(1):24–45, 1998.

[66] B. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: An introduction to TCOZ. In K. Futatsugi, R. Kemmerer, and K. Torii, editors, *The 20th International Conference on Software Engineering (ICSE'98)*, pages 95–104. IEEE Press, 1998.

[67] B. Mahony and J. S. Dong. Network Topology and a Case Study in TCOZ. In J. Bowen, A. Fett, and M. Hinchey, editors, *ZUM'98: The 11th International Conference of Z Users*, volume 1493 of *Lecture Notes in Computer Science*, pages 308–327, Berlin, Germany, September 1998. Springer-Verlag.

[68] B. Mahony and J. S. Dong. Overview of the semantics of tcoz. In *IFM '99: Proceedings of the 1st International Conference on Integrated Formal Methods*, pages 66–85, London, UK, 1999. Springer-Verlag.

[69] B. Mahony and J. S. Dong. Sensors and Actuators in TCOZ. In *FM '99: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume II*, pages 1166–1185, London, UK, 1999. Springer-Verlag.

[70] B. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, February 2000.

[71] A. P. Mathur and W. E. Wong. An empirical comparison of data flow and mutation-based test adequacy criteria. *Software Testing, Verification and Reliability*, 4(1):9–31, 1994.

[72] A. P. Mathur and W. E. Wong. A theoretical comparison between mutation and data flow based test adequacy criteria. In *Proceedings of the 22nd Annual ACM Computer Science Conference*, pages 38–45, 1994.

[73] A. M. Memon and M. L. Soffa. Regression testing of GUIs. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 118–127, 2003.

[74] T. Mens, J. Buckley, M. Zenger, and A. Rashid. Towards a taxonomy of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice (Special Issue on USE)*, 17(5), 2005.

[75] T. Miller and P. Strooper. A framework for systematic specification animation. Technical Report 02-35, The University of Queensland, 2000.

[76] T. Miller and P. Strooper. Model-based specification animation using test-graphs. In *ICFEM '02: Proceedings of the 4th International Conference on Formal Engineering Methods*, pages 192–203. Springer-Verlag, 2002.

[77] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[78] A. J. Offutt, Y. W. Xiong, and S. Y. Liu. Criteria for generating specificaiton-based tests. In *Proceedings of the fifth International Conference on Engineering of Complex Computer Systems*, pages 119–131, Las Vegas, 1999. IEEE Computer Society Press.

[79] J. Offutt, S. Y. Liu, A. Abdurazik, and P. Ammann. Generating test data from state-based specifications. *Software Testing, Verification and Reliability*, 13:25–53, 2003.

[80] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *ECOOP'05 - Proceedings of the 19th European Object-Oriented Programming*, pages 214–240, 2005.

[81] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, 1985.

[82] G. Rothermel and M. J. Harrold. Selecting regression tests for object-oriented software. In *Proceedings of the Conference on Software Maintenance*, pages 14–25, 1994.

[83] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, August 1996.

[84] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.

[85] G. Rothermel, M. J. Harrold, and J. Dedhia. Regression test selection for C++ software. *Journal of Software Testing, Verification, and Reliability*, 10(6):77–109, June 2000.

[86] A. Sampaio, R. Chitchyan, A. Rashid, and P. Rayson. Ea-miner: a tool for automating aspect-oriented requirements identification. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 352–355, 2005.

[87] I. Sanabria-Piretti. *Data Refinement by Rewriting*. PhD thesis, Department of computer science, Oxford University, 2001.

[88] M. Satpathy, M. Leuschel, and M. J. Butler. ProTest: An automatic test environment for B specifications. *Electronic Notes Theoretical Computer Science*, 111:113–136, 2005.

[89] S. Schneider and J. Davies. A brief history of Timed CSP. *Theoretical Computer Science*, 138:243–271, 1995.

[90] S. Schneider, J. Davies, D. M. Jackson, G. M. Reed, J. N. Reed, and A. W. Roscoe. Timed CSP: Theory and practice. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, pages 640–675. Springer-Verlag, 1992.

[91] B. A. Schroeder. On-line monitoring: A tutoiral. *IEEE Computer*, 28(6):72–78, 1995.

[92] G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.

[93] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1989.

[94] D. Stein, S. Hanenberg, and R. Unland. A UML-based aspect-oriented design notation for AspectJ. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 106–112, 2002.

[95] D. Stein, S. Hanenberg, and R. Unland. Expressing different conceptual models of join point selections in aspect-oriented design. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 15–26, 2006.

[96] M. Sulzmann and M. Wang. Aspect-oriented programming with type classes. In *FOAL '07: Proceedings of the 6th workshop on Foundations of aspect-oriented languages*, pages 65–74, 2007.

[97] J. Sun, J. S. Dong, J. Liu, and H. Wang. A XML/XSL Approach to Visualize and Animate TCOZ. In *The 8th Asia-Pacific Software Engineering Conference (APSEC'01)*, pages 453–460. IEEE Press, 2001.

[98] J. Suzuki and Y. Yamamoto. Extending UML with aspect: Aspect support in the desgin phase. In *Proceedings of the 3rd AOP workshop at ECOOP'99*, 1999.

[99] P. Tarr and H. Ossher. *Hyper/J User and Installation Manual*. IBM Corporation, 2000.

[100] P. Tarr, H. Ossher, W. H. Harrison, and S. M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.

[101] F. Tessier, L. Badri, and M. Badri. A model-based detection of semantic conflicts between aspects: Towards a formal approach. In *Proceedings of International Workshop on Aspect-Oriented Software Development*, 2004.

[102] W. E. Truszkowski, M. G. Hinchey, J. L. Rash, and C. A. Rouff. NASA's swarm missions: The challenge of building autonomous software. *IT Professional*, 6(5):47–52, 2004.

[103] W. E. Truszkowski, M. G. Hinchey, J. L. Rash, and C. A. Rouff. Autonomous and autonomic systems: A paradigm for future space exploration mission. *IEEE Transactions on Systems, Man and Cybermetrics, Part C: Applications and Reviews*, 36(3):279–291, 2006.

[104] N. Ubayashi and S. Nakajima. Context-aware feature-oriented modeling with an aspect extension of VDM. In *Proceedings of 22nd Annual ACM Symposium on Applied Computing*, 2007.

[105] M. Utting. Data structures for Z testing tools. In *Proceedings of FM-TOOLS*, 2000.

[106] F. Vokolos and P. Frankl. Pythia: A regression test selection tool based on textual differencing. In *International Conference on Reliability, Qaulity, and Safety of Software Intensive Systems*, May 1997.

[107] M. Wang, K. Chen, and S. C. Khoo. On the pursuit of staticness and coherence. In *Proceedings of the 5th Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, 2006.

[108] M. Wang, K. Chen, and S. C. Khoo. Type-directed weaving of aspects for higher-order functional languages. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 78–87, 2006.

[109] L. J. White and K. Abdullah. A firewall approach for regression testing of object-oriented software. In *Proceedings of 10th Annual Software Quality Week*, May 1997.

[110] L. J. White and H. K. N. Leung. A firewall concept for both control-flow and data-flow in regression integration testing. In *Proceedings of the conference on Software Maintenance '92*, pages 262–270, November 1992.

[111] W. E. Wong, J. R. Horgan, S. London, and H. A. Bellcore. A study of effective regression testing in practice. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering (ISSRE '97)*, pages 522–528, 1997.

[112] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof.* Prentice-Hall International, 1996.

[113] D. X. Xu and K. E. Nygard. Threat-driven modeling and verification of secure software using aspect-oriented petri nets. *IEEE Transactions on Software Engineering*, 32(4):265–278, 2006.

[114] H. Yu, D. Liu, Z. Shao, and X. He. Modeling complex software systems using an aspect extension of Object-Z. In *Proceedings of 18th International Conference on Software Engineering and Knowledge Engineering, 2006*, pages 11–16, 2006.

[115] H. Yu, D. Liu, L. Yang, and X. He. Formal aspect-oriented modeling and analysis by AspectZ. In *Proceedings of 17th International Conference on Software Engineering and Knowledge Engineering, 2005*, pages 175–180, 2005.

[116] J. Zhao and M. Rinard. Pipa: A behavioral interface specification language for AspectJ. In *Proceedings of Fundamental Approaches to Software Engineering (FASE)*, pages 150–165, 2003.

[117] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.

Appendix A

Specification of Railway Control System in Z Notation

$$SegmentState ::= Free \mid Occupied$$
$$SignalState ::= Red \mid Green \mid Yellow \mid Off$$
$$TrackState ::= OpenAB \mid OpenBA \mid Closed$$
$$TrainDirection ::= MoveAB \mid MoveBA \mid Stop$$
$$TrainPosition ::= A \mid B \mid OffTrack$$

Segment

$$status : SegmentState;\ sigAB, sigBA : SignalState$$

$$sigAB \neq Off \Rightarrow sigBA = Off$$
$$sigBA \neq Off \Rightarrow sigAB = Off$$

Train

$$pos : TrainPosition$$
$$dir : TrainDirection$$

Track

$$status : TrackState;\ segA, segB : Segment;\ trnOne, trnTwo : Train$$

$$status = OpenAB \land segA.status = Free = segB.status \Rightarrow segA.sigAB = Green$$
$$status = OpenAB \land segA.status = Occupied \Rightarrow segA.sigAB = Red$$
$$status = OpenAB \land segA.status = Free \land segB.status = Occupied \Rightarrow segA.sigAB = Yellow$$
$$status = OpenBA \land segB.status = Occupied \Rightarrow segB.sigBA = Red$$
$$status = OpenBA \land segA.status = Free = segB.status \Rightarrow segB.sigBA = Green$$
$$status = OpenBA \land segB.status = Free \land segA.status = Occupied \Rightarrow segB.sigBA = Yellow$$
$$status = Closed \Rightarrow segA.sigAB = Off = segA.sigBA \land segB.sigAB = Off = segB.sigBA$$
$$trnOne.pos \neq OffTrack \land trnTwo.pos \neq OffTrack \Rightarrow trnOne.pos \neq trnTwo.pos$$

InitTrack

$Track'$

$status' = OpenAB \land segA'.status = Free = segB'.status$
$segA'.sigAB = Green = segB'.sigAB$
$trnOne'.dir = MoveAB = trnTwo'.dir$
$trnOne'.pos = OffTrack = trnTwo'.pos$

trnOneEnter

$\Delta Track$

$trnOne.pos = OffTrack \land status = OpenAB \land trnOne.dir = MoveAB$
$segA.sigAB = Green \lor segA.sigAB = Yellow$
$segA'.status = Occupied \land segA'.sigAB = Red$
$trnOne'.pos = A \land trnOne'.dir = trnOne.dir$
$trnTwo' = trnTwo \land status' = status \land segB' = segB$

trnTwoEnter

$\Delta Track$

$trnTwo.pos = OffTrack \land status = OpenAB \land trnTwo.dir = MoveAB$
$segA.sigAB = Green \lor segA.sigAB = Yellow$
$segA'.status = Occupied \land segA'.sigAB = Red$
$trnTwo'.pos = A \land trnTwo'.dir = trnTwo.dir$
$trnOne' = trnOne \land status' = status \land segB' = segB$

trnOneMovetoB

$\Delta Track$

$trnOne.pos = A \land status = OpenAB \land trnOne.dir = MoveAB$
$segB.sigAB = Green \land segA'.status = Free$
$segA'.sigAB = Yellow \land segB'.sigAB = Red \land segB'.status = Occupied$
$trnOne'.pos = B \land trnOne'.dir = trnOne.dir$
$trnTwo' = trnTwo \land status' = status$

trnTwoMovetoB

$\Delta Track$

$trnTwo.pos = A \land status = OpenAB \land trnTwo.dir = MoveAB$
$segB.sigAB = Green \land segA'.status = Free$
$segA'.sigAB = Yellow \land segB'.sigAB = Red \land segB'.status = Occupied$
$trnTwo'.pos = B \land trnTwo'.dir = trnOne.dir$
$trnOne' = trnOne \land status' = status$

Appendix B

Implementation of Railway Control System in Java

```
  import java.util.*;
  import java.io.*;
  enum SegState {Free,Occupied}
  enum SigState {Red, Green, Yellow, Off}
  enum TrackState {OpenAB,OpenBA, Closed}
  enum rainDirection{MoveAB,MoveBA, Stop}
  enum TrainPosition{A,B,OffTrack}
class Track {
     TrackState status;
     Segment segmentA, segmentB;
     Train trainOne, trainTwo;
     Track(){
         status = TrackState.OpenAB;
         segmentA = new Segment();
         segmentB = new Segment();
         trainOne = new Train();
         trainTwo = new Train(); }
     public void oneEnter(){
        if(status == TrackState.OpenAB
             &&trainOne.position == TrainPosition.OffTrack
              && trainOne.direction == TrainDirection.MoveAB
               && (segmentA.signalAB == SigState.Green ||
                           segmentA.signalAB == SigState.Yellow))
              {  segmentA.status = SegState.Occupied;
                 segmentA.signalAB = SigState.Red;
                 trainOne.position = TrainPosition.A; }}
     public void twoEnter(){
        if(status == TrackState.OpenAB
             && trainTwo.position == TrainPosition.OffTrack
              && trainTwo.direction == TrainDirection.MoveAB
               && (segmentA.signalAB == SigState.Green ||
                           segmentA.signalAB == SigState.Yellow))
              {  segmentA.status = SegState.Occupied;
                 segmentA.signalAB = SigState.Red;
                 trainTwo.position = TrainPosition.A; }}
     public void oneAtoB(){
```

```
        if(status == TrackState.OpenAB
              && trainOne.position == TrainPosition.A
               && trainOne.direction == TrainDirection.MoveAB
                && segmentB.signalAB == SigState.Red )
              { segmentA.status = SegState.Free;
                segmentA.signalAB = SigState.Yellow;
                segmentB.signalAB = SigState.Green;
                segmentB.status = SegState.Occupied;
                trainOne.position = TrainPosition.B; }}
    public void twoAtoB(){
        if(status == TrackState.OpenAB
              && trainTwo.position == TrainPosition.A
               && trainTwo.direction == TrainDirection.MoveAB)
              { segmentA.status = SegState.Free;
                segmentA.signalAB = SigState.Yellow;
                segmentB.signalAB = SigState.Red;
                segmentB.status = SegState.Occupied;
                trainTwo.position = TrainPosition.B; }}
           }
class Segment{
    SegState status;
    SigState signalAB, signalBA;
    Segment(){ status = SegState.Free;
                signalAB = SigState.Green;
                signalBA = SigState.Off; }
           }
class Train{
    TrainPosition position;
    TrainDirection direction;
    Train(){ position = TrainPosition.OffTrack;
             direction = TrainDirection.MoveAB; }
           }
```

Appendix C

Specification of Robotic Assembly System in Z Notation

$$Part ::= PowerSys \mid NavigationSys \mid ControlSys \mid CommunicationSys \mid MagnetoMeter \mid SpectroMeter$$

RobotSystem

$$leftarm, rightarm : \operatorname{seq} Part$$
$$tempstack : \operatorname{seq} Part$$
$$currentproduct : Part \pfun \mathbb{N}$$

InitRobotSystem

$$RobotSystem'$$

$$leftarm' = \langle\rangle \land rightarm' = \langle\rangle \land tempstack' = \langle\rangle$$
$$currentproduct' = \varnothing$$

LeftArmPick

$$\Delta RobotSystem$$
$$part? : Part$$

$$tempstack = \langle\rangle \lor head\ tempstack \in \operatorname{dom} currentproduct \lor$$
$$head\ tempstack \notin \{PowerSys, NavigationSys, ControlSys\}$$
$$part? \in \{PowerSys, NavigationSys, ControlSys\}$$
$$leftarm = \langle\rangle \land leftarm' = \langle part?\rangle$$
$$tempstack' = tempstack \land rightarm' = rightarm$$
$$currentproduct' = currentproduct$$

LeftArmGetFromStack

$\Delta RobotSystem$

$\#tempstack > 0 \land \#leftarm = 0$
$head\ tempstack \notin \mathrm{dom}\ currentproduct$
$head\ tempstack \in \{PowerSys, NavigationSys, ControlSys\}$
$leftarm' = \langle head\ tempstack \rangle \land rightarm' = rightarm$
$currentproduct' = currentproduct \land tempstack' = tail\ tempstack$

LeftArmRelease

$\Delta RobotSystem$
$part! : Part$

$\#leftarm = 1 \land leftarm(1) \notin \mathrm{dom}\ currentproduct$
$part! = leftarm(1) \land leftarm' = \langle\rangle$
$leftarm(1) = PowerSys \Rightarrow$
$currentproduct' = currentproduct \cup \{part! \mapsto 1\}$
$leftarm(1) = NavigationSys \Rightarrow$
$currentproduct' = currentproduct \cup \{part! \mapsto 2\}$
$leftarm(1) = ControlSys \Rightarrow$
$currentproduct' = currentproduct \cup \{part! \mapsto 3\}$
$tempstack' = tempstack \land rightarm' = rightarm$

LeftArmPushToStack

$\Delta RobotSystem$
$parttopush! : Part$

$\#leftarm = 1 \land leftarm(1) \in \mathrm{dom}\ currentproduct$
$parttopush! = leftarm(1) \land leftarm' = \langle\rangle$
$tempstack' = \langle parttopush! \rangle \frown tempstack$
$rightarm' = rightarm \land currentproduct' = currentproduct$

RightArmPick

$\Delta RobotSystem$
$part? : Part$

$tempstack = \langle\rangle \lor head\ tempstack \in \mathrm{dom}\ currentproduct \lor$
$head\ tempstack \notin \{CommunicationSys, SpectroMeter, MagnetoMeter\} \lor$
$\mathrm{dom}(currentproduct \rhd \{5\}) \cup \{head\ tempstack\} =$
$\{SpectroMeter, MagnetoMeter\}$
$part? \in \{CommunicationSys, SpectroMeter, MagnetoMeter\}$
$rightarm = \langle\rangle \land rightarm' = \langle part? \rangle \land tempstack' = tempstack$
$currentproduct' = currentproduct \land leftarm' = leftarm$

RightArmGetFromStack

$\Delta RobotSystem$

$\#tempstack > 0 \land \#rightarm = 0$

$head\ tempstack \notin \mathrm{dom}\ currentproduct$

$\mathrm{dom}(currentproduct \rhd \{5\}) \cup \{head\ tempstack\} \neq \{SpectroMeter, MagnetoMeter\}$

$head\ tempstack \in \{CommunicationSys, SpectroMeter, MagnetoMeter\}$

$rightarm' = \langle head\ tempstack \rangle \land leftarm' = leftarm$

$currentproduct' = currentproduct \land tempstack' = tail\ tempstack$

RightArmRelease

$\Delta RobotSystem$

$part! : Part$

$\#rightarm = 1 \land rightarm(1) \notin \mathrm{dom}\ currentproduct$

$\mathrm{dom}(currentproduct \rhd \{5\}) \cup \{rightarm(1)\} \neq \{SpectroMeter, MagnetoMeter\}$

$part! = rightarm(1)$

$rightarm(1) = CommunicationSys \Rightarrow currentproduct' = currentproduct \cup \{part! \mapsto 4\}$

$(rightarm(1) = MagnetoMeter \lor rightarm(1) = SpectroMeter) \Rightarrow currentproduct' = currentproduct \cup \{part! \mapsto 5\}$

$rightarm' = \langle\rangle \land tempstack' = tempstack \land leftarm' = leftarm$

RightArmPushToStack

$\Delta RobotSystem$

$parttopush! : Part$

$\#rightarm = 1 \land (rightarm(1) \in \mathrm{dom}\ currentproduct \lor \mathrm{dom}(currentproduct \rhd \{5\}) \cup \{rightarm(1)\} = \{SpectroMeter, MagnetoMeter\})$

$parttopush! = rightarm(1) \land rightarm' = \langle\rangle$

$tempstack' = \langle parttopush! \rangle \frown tempstack$

$leftarm' = leftarm \land currentproduct' = currentproduct$

ReleaseProduct

$\Delta RobotSystem$

$producttorelease! : Part \pfun \mathbb{N}$

$\#currentproduct = 5$

$currentproduct(PowerSys) = 1$

$currentproduct(NavigationSys) = 2$

$currentproduct(ControlSys) = 3$

$currentproduct(CommunicationSys) = 4$

$(currentproduct(MagnetoMeter) = 5 \lor currentproduct(SpectroMeter) = 5)$

$producttorelease! = currentproduct \land currentproduct' = \varnothing$

$leftarm' = \langle\rangle \land rightarm' = \langle\rangle \land tempstack' = tempstack$

Appendix D

Implementation of Robotic Assembly System in Java

```
import java.util.*;
public class RobotSystem {
    Vector<Object> leftarm, rightarm;
    Stack<Object> tempstack;
    Vector<Object> currentproduct;
    Vector<Object> partsToLeft;
    Vector<Object> partsToRight;
RobotSystem() {
    leftarm = new Vector<Object>();
    rightarm = new Vector<Object>();
    tempstack = new Stack<Object>();
    currentproduct = new Vector<Object>();
    for (int index = 0; index < 5; index++)
    { currentproduct.addElement(""); }
    partsToLeft = new Vector<Object>();
    partsToLeft.addElement("PowerSys");
    partsToLeft.addElement("NavigationSys");
    partsToLeft.addElement("ControlSys");
    partsToRight = new Vector<Object>();
    partsToRight.addElement("CommunicationSys");
    partsToRight.addElement("MagnetoMeter");
    partsToRight.addElement("SpectroMeter");
}
public void leftArmPick(Object lPart) {
    if (leftarm.isEmpty())
    {
        if(lPart == "PowerSys" || lPart == "NavigationSys" ||
                                            lPart == "ControlSys")
        {
          if (!tempstack.empty())
              {
              Object topOfTempstack = tempstack.peek();
              if (currentproduct.contains(topOfTempstack) ||
                          !partsToLeft.contains(topOfTempstack))
                  { leftarm.addElement(lPart); }
```

```
            }
          else
              { leftarm.addElement(lPart); }
        }
    }
}
public void leftArmGetFromStack() {
    if(!tempstack.empty() && leftarm.isEmpty())
    {
        Object topOfTempstack = tempstack.peek();
        if(!currentproduct.contains(topOfTempstack) &&
                            partsToLeft.contains(topOfTempstack))
        {
            leftarm.addElement(topOfTempstack);
            topOfTempstack = tempstack.pop();
        }
    }
}
public void leftArmRelease() {
    if (!leftarm.isEmpty())
    {
        Object releasedPart = leftarm.get(0);
        if (!currentproduct.contains(releasedPart))
        {
            if (releasedPart == "PowerSys")
            {currentproduct.setElementAt(releasedPart,0);}
            else if (releasedPart == "NavigationSys")
            {currentproduct.setElementAt(releasedPart,3);}
            else if (releasedPart == "ControlSys")
            {currentproduct.setElementAt(releasedPart,2);}
            leftarm.clear();
        }
    }
}
public void leftArmPushToStack() {
    if(!leftarm.isEmpty())
    {
        Object itemInRight = leftarm.get(0);
        if(currentproduct.contains(itemInRight))
        {
            tempstack.push(itemInRight);
            leftarm.clear();
        }
    }
}
public void rightArmPick(Object rPart) {
    if (rightarm.isEmpty())
    {
        if(rPart == "CommunicationSys" || rPart == "MagnetoMeter"
                                       || rPart == "SpectroMeter")
        {
           if (!tempstack.empty())
           {
            Object topOfTempstack = tempstack.peek();
            if (currentproduct.contains(topOfTempstack)||
                        !partsToRight.contains(topOfTempstack))
               { rightarm.addElement(rPart); }
```

```
            else if ((topOfTempstack == "MagnetoMeter" &&
                currentproduct.lastElement() == "SpectroMeter")||
                (topOfTempstack == "SpectroMeter" &&
                 currentproduct.lastElement() == "MagnetoMeter"))
               { rightarm.addElement(rPart); }
           }
           else
           {  rightarm.addElement(rPart); }
        }
    }
}
public void rightArmGetFromStack() {
    if(!tempstack.empty() && rightarm.isEmpty())
    {
        Object topOfTempstack = tempstack.peek();
        if(!currentproduct.contains(topOfTempstack)&&
                          partsToRight.contains(topOfTempstack))
        {
                rightarm.addElement(topOfTempstack);
                topOfTempstack = tempstack.pop();
        }
    }
}
public void rightArmRelease() {
    if(!rightarm.isEmpty())
    {
        Object releasedPart = rightarm.get(0);
        if (!currentproduct.contains(releasedPart))
        {
            if(releasedPart == "CommunicationSys")
                {currentproduct.setElementAt(releasedPart, 1);}
            else if (releasedPart == "MagnetoMeter"||
                              releasedPart == "SpectroMeter")
                {currentproduct.setElementAt(releasedPart, 4);}
            rightarm.clear();
        }
    }
}
public void rightArmPushToStack() {
    if(!rightarm.isEmpty())
    {
        Object itemInLeft = rightarm.get(0);
        if(currentproduct.contains(itemInLeft)||(itemInLeft ==
            "MagnetoMeter" && currentproduct.lastElement() ==
              "SpectroMeter") || (itemInLeft == "SpectroMeter" &&
                 currentproduct.lastElement() == "MagnetoMeter"))
        {
         tempstack.push(itemInLeft);
         rightarm.clear();
         }
   }
}
```

```
public Vector<Object> releaseProduct() {
    Vector<Object> product = new Vector<Object>();
    product = currentproduct;
    currentproduct.clear();
    currentproduct = new Vector<Object>();
    for (int index = 0; index < 5; index++)
    { currentproduct.addElement(""); }
    return product;
 }
}
```